覆盖设计、原型、开发、上架完整知识
iOS 设计/开发疑难问题图文解析
从 0 到 1 做出属于自己的 iOS 应用

Broadview®
www.broadview.com.cn

自己动手做 iOS App

从设计开发到上架 App Store

张子怡 著

Sketch 44 + iOS 10 + Swift 3.1

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

献给我的父母和妻子同同

爱是一切作品的根源

自己动手做 iOS App

从设计开发到上架 App Store

张子怡 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书为想要接触 iOS 应用设计、开发的读者提供了由浅入深的详细指导。从 iOS 应用制作的步骤是什么，应该使用什么软件，如何发布应用到 App Store，到 iOS 的设计理念是什么，如何正确书写 Swift 语言，再到后端和客户端是如何交互运作的等，本书配合图示，精辟、直观地阐明了 iOS 应用制作中的种种疑问。

如果你是一位第一次接触 iOS 应用制作的新手，那么读完本书你将会充满信心地着手把自己的想法带到现实。即使是有一定经验的 iOS 设计师也可以学到如何编写代码，程序员则可以学习到设计知识，并都能够获得独立完成个人作品的能力。

图书在版编目（CIP）数据

自己动手做 iOS App：从设计开发到上架 App Store / 张子怡著. —北京：电子工业出版社，2017.8
ISBN 978-7-121-32019-4

Ⅰ. ①自… Ⅱ. ①张… Ⅲ. ①移动终端－应用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字（2017）第 140710 号

责任编辑：黄爱萍
印　　刷：北京千鹤印刷有限公司
装　　订：北京千鹤印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：720×1000　1/16　印张：11.5　字数：140 千字
版　　次：2017 年 8 月第 1 版
印　　次：2017 年 8 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

「前　言」

关于这本书

这几年 Apple 产品的迭代速度变快，连带着对应用制作需要学习的知识更新也加快了。连有经验的制作者都感叹今天学习的知识很快就用不上了，对于新人来说，恐怕更不知道如何下手接触应用制作。但是即使知识更迭得再快，只要沉淀下基本的构建思想和制作过程，便能以不变应万变，一步步深入地学习应用制作技能。

我喜欢把做软件应用比作设计制造一辆汽车，有各种专业书籍对怎么设计车的外型、用什么材料、发动机应该怎么组装、以及车内该如何布置等详细讲解，但是对于一个新人来说，这些书籍摆在他面前，他也无法知道从何开始，即使把发动机部分学得滚瓜烂熟，去实际生产一辆车恐怕也是障碍重重。本书将需要用到什么工具、如何设计、如何开发、如何发布到市场等知识贯穿起来，手把手地带领新人从零开始，我想从这个角度切入 iOS 应用制作，来得更实际一些。

2016 年年底我设计、开发并上架了一款名为 Oslo 的 iOS 应用，可以在 https://itunes.apple.com/us/app/oslo-mobile-unsplash/id1184226442?mt=8，或者在 App Store 中搜索 Oslo Mobile Unsplash 下载。这款应用的 UI 设计是 Sketch，应用图标设计是 Affinity Designer，开发环境是 Xcode 8.2.1，开发语言是 Swift 3.0.2。我相信边学习、边实践是最快、最有效的途径，因此这本书会以这款应用为案例，带你一步一步制作一款最终上架到 App Store 的应用。学习过程中不但涉及设计或者开发方面的概念讲解，同时还结合了实际的制作，让你对 iOS 的应用制作有更深刻的了解。

当你跟随这本书全部做下来后，你将会充满信心地做出属于自己的 iOS 应用，当初那种对设计缺乏自信，对编程敬而远之的心情将不复存在。所以，上路吧！

谁需要读这本书

如果你是一名没有接触过 iOS 应用制作的新人，你是否有过这样的问题：

- 我该从哪里入手学习 iOS 应用制作，哪些资源是针对初学者的，又如何入门？
- 要使用哪些工具，如何快速上手？

……

如果你是一名设计师，你是否有过这样的问题：

- 我应该用什么规格的画布来做 UI，应该导出什么样尺寸的图标应用到开发中？
- 怎样才能快速填充好各种头像？
- 怎样做快速原型（Fast Prototype）？
- 看到代码就害怕，即使学习了也无法应用到实际中，该怎么办？

……

如果你是一名工程师，你是否有过这样的问题：

- 怎样通过 Storyboard 快速、简单地实现多屏幕适配？
- 对色彩和绘图完全不在行，这样如何做设计？
- 我想了解一些最新的语法和开发环境的特性。

……

所以无论是入门者或专业人士，都可以从本书吸取到适合自己的新的知识，或者对平常模棱两可的问题得到答案。当然如果你只是初次接触 iOS 应用制作，那么这本书会由浅入深，一步一步带你拓展和提升，最终获得不输给专业人士的能力。同时，本书适合有热情制作 iOS 应用的所有人。

使用到的工具

本书使用到的硬件有 MacBook、iMac、Mac mini 三种，使用系统为 macOS。

Xcode 8.3.3

Xcode 是 Apple 系列产品开发的主要工具，同时包含了 Swift。在 Mac App Store 中搜索 Xcode 下载，或者在 https://developer.apple.com/download/ 下载测试版。书中使用的版本是 Xcode 8.3.3。注意，Xcode 8.0 才包含 Swift 3，这也是书中主要使用的程序语言。

Sketch 44.1

Sketch 帮助快速实现应用原型，也能满足界面要求较高的制作。在 http://sketchapp.com/ 下载。书中使用的版本是 Sketch 44.1。

Swift 3.1

Swift 为书中使用的编程语言。同时也是 Apple 系列产品开发的主要编程语言。Swift 还在不断演化过程中，可以在 https://swift.org/ 了解学习。正式版 Xcode 包含了最新的稳定版 Swift，如果想体验测试版的 Swift，可以下载测试版 Xcode。书中所使用的版本是 3.1。

本书主要使用到的工具就是以上这些，在学习过程中需要使用到其他软件时会单独说明。

设计资源和源代码

在章节讲解的过程中，会指出资源的下载地址。这些资源能够自由用于个人或商业用途，用于公开演说或者教育用途时，希望能够注明来源。

源代码可以在 https://github.com/hipposan/Oslo 获取，但不能用于出售或其他交易。

随着 Sketch、Xcode 和 Swift 的不断更新换代，对于可能出现的制作方式及语法的更新，我也会尽自己所能更新本书，同时更新下载地址中的资源。

勘误和反馈

对于书中出现的任何错误，或者在使用过程中有不明白的地方，可以发送邮件到 zzy0600@gmail.com，我会认真查看每一封信件，希望能和大家多多交流。

「目　录」

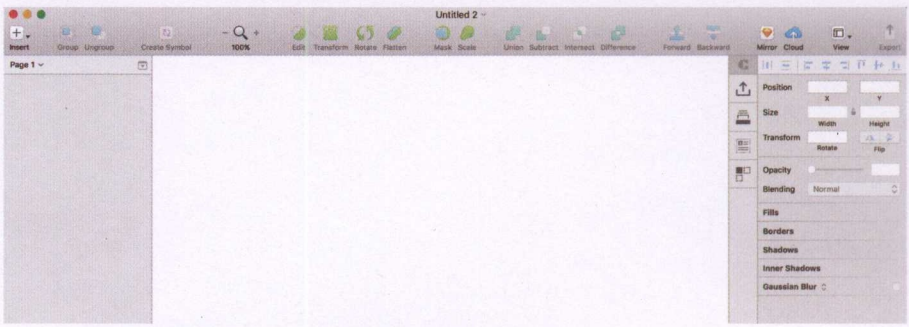
从现在开始，我们将会从 0 到 1，设计并开发一款名为 Oslo 的 Unsplash[1] 第三方客户端。在这个过程中边做边学习设计理念、工具使用、程序编写等知识。如果你做好了准备，那我们就开始吧！

Sketch 介绍

早些年大家做设计时基本都用 Photoshop。但 Photoshop 毕竟是一款作图软件，在做 Web/App UI 设计，尤其是原型设计时，通常不需要用到精细的图片打磨或者多样的笔刷，反而是合适的画布选择和方便的资源（assets）导出等是首选需要。因此对于新手来说，打开 Photoshop 后要先去了解大量的非常用功能，让人十分头疼。

由于这个原因，Sketch 出现了。Sketch 在作图方面与 Phtoshop 的定位不同，Sketch 定位于 Web/App UI 设计，因此对于这部分设计人员来说更易上手，使用起来也更加直接，并且完全是矢量图，于是很多设计师马上从 Photoshop 转移到了 Sketch 阵地。

安装好 Sketch 后，请打开 Sketch，选择 New Document，展现的界面如下图所示。



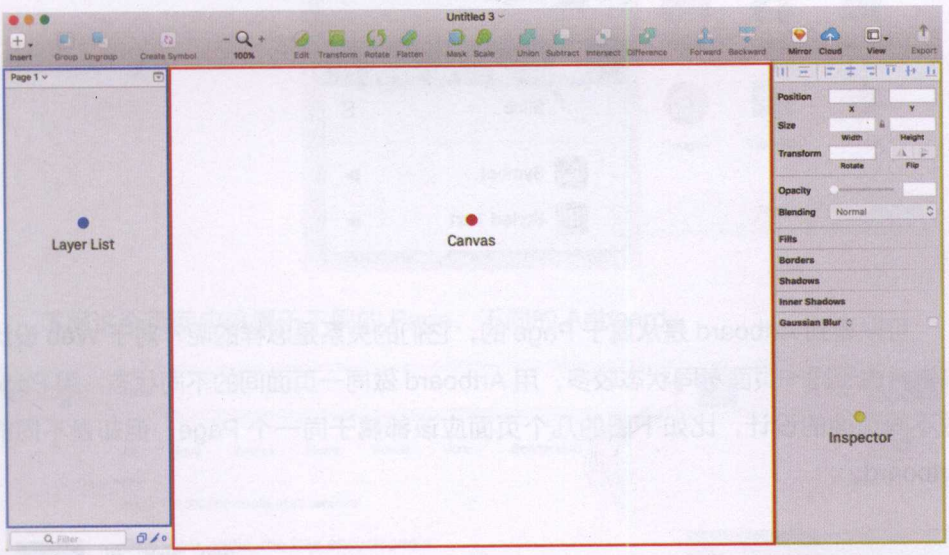
首先是顶部的 Toolbar[2]，如下图所示。

1 Unsplash 是一个高清摄影的分享平台，里面的图片可以在个人或商业作品中自由使用。

2 由于使用的工具为英文版，在网上搜索的资料也对应为英文，因此之后的专业名词大多以英文出现。

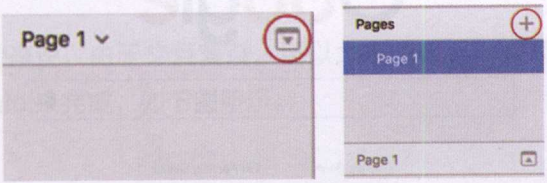
工具栏主要是一些常用的快捷操作，可以通过单击右键来进行自定义。在实际操作过程中，用顶部菜单、右键菜单或者快捷键也能达到相应目的。

接下来的三个区域是在实际项目中最为常用的。由左开始，分别是 Layer List，Canvas 和 Inspector，如下图所示。

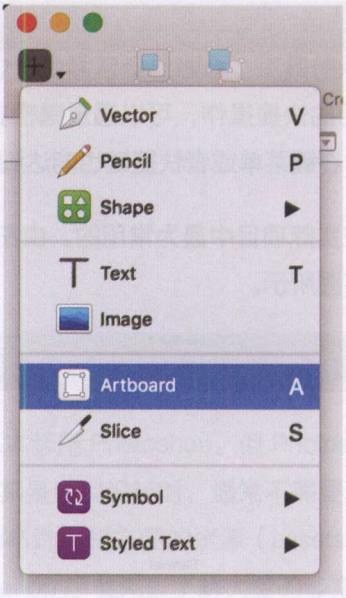


Layer List 是图层的结构，类似于文件目录。这里需要注意的是 Page 和 Artboard。

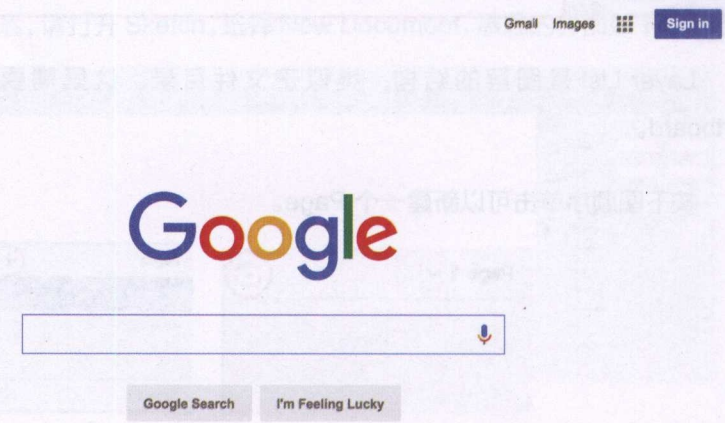
按下图顺序单击可以新建一个 Page。

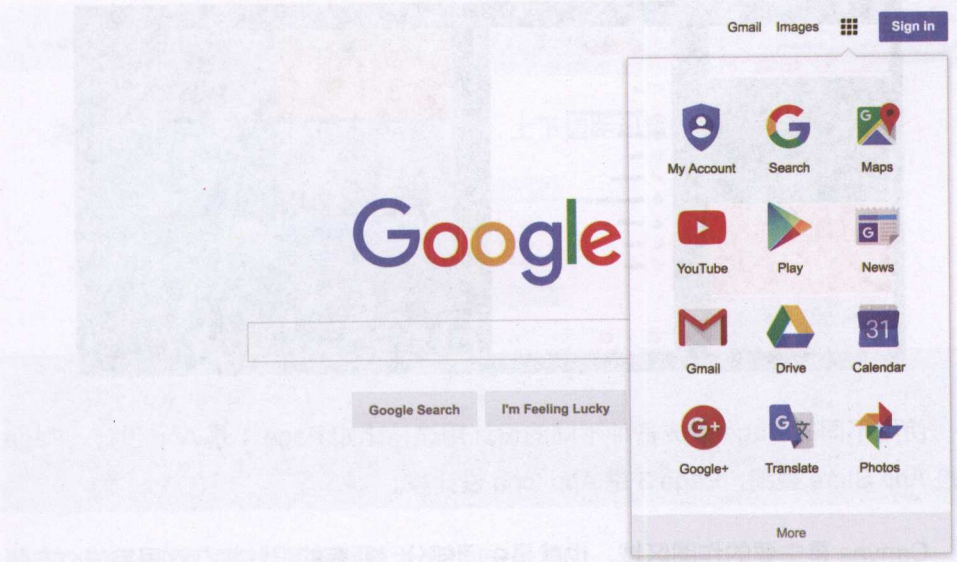


单击下图中的 Artboard，然后在中间 Canvas 区域随意拖曳会新建一个 Artboard。

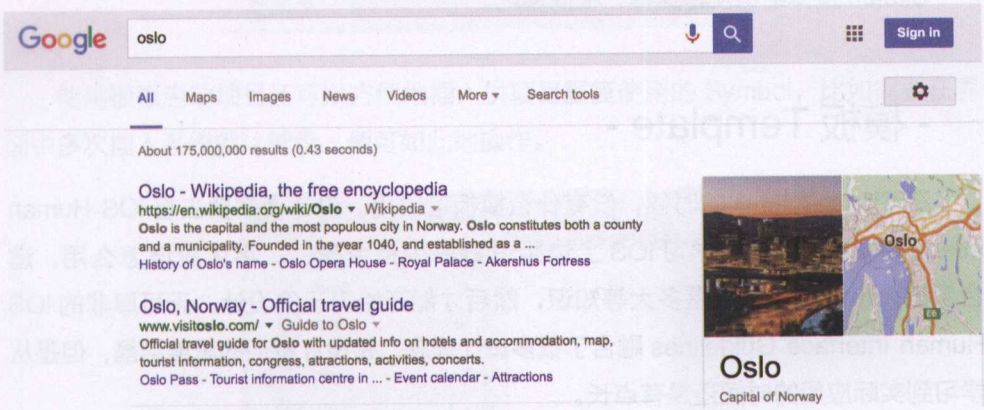


可以看到 Artboard 是从属于 Page 的，它们的关系是怎样的呢？对于 Web 设计来说，由于同一页面不同状态较多，用 Artboard 做同一页面间的不同状态，用 Page 做不同页面的设计，比如下图的几个页面应该都属于同一个 Page，但却是不同的 Artboard。

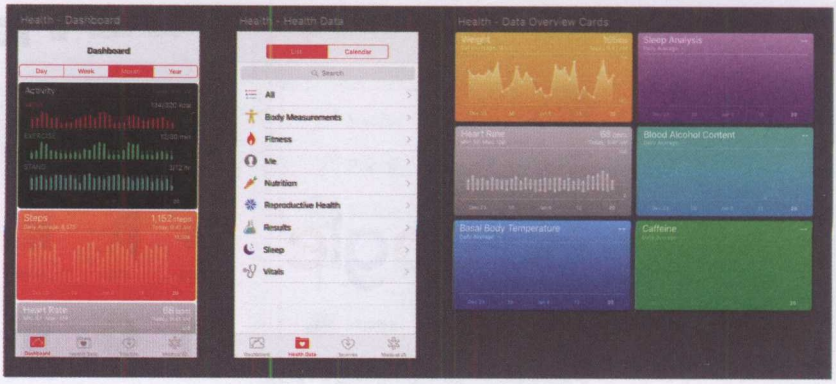




下面这个页面应该属于不同的 Page，不同的 Artboard。



对于 App 设计来说，由于交互复杂，所以大多数情况下都在一个 Page 内，而使用不同的 Artboard 来完成，如下图所示。



使用不同的 Page 完成各种不同的设计用途，比如 Page 1 是 App 设计，Page 2 是 App Store 截图，Page 3 是 App Icon 设计等。

Canvas 是主要的作图区域，也就是中间部分，所有的设计将在这里完成。右侧的 Inspector 用来调整元素属性。

Sketch 整体结构就是这样，很简单吧。

- 模板 Template -

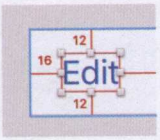
在 Sketch 刚推出的时候，没有什么模板可以用。设计者需要去看 iOS Human Interface Guidelines，学习 iOS 上边距设置多少是合理的，字体应该怎么用，适合手指大小的单击区域是多大等知识，然后才能开始自己的设计。无可厚非的 iOS Human Interface Guidelines 融合了很多设计理念，是设计者们的思维结晶，但是从学习到实际应用的时间还是有点长。

模板为设计者提供了更快捷的学习方法。比如 Great Simple Studio 推出的 iOS 10 GUI（http://ios10.greatsimple.io/）。

这些模板都遵照 iOS Human Interface Guidelines 的标准规范，并且附带很多原生的 iOS 界面。我们可以从里面直接可视化地学到很多实用知识，以 Great Simple Studio 的 iOS 10 GUI 为例，下载 iOS10 GUI 后打开 Sketch 文件，具体如下。

比如 iOS 的状态栏（Status bar）设置宽度为 375，高度为 20。

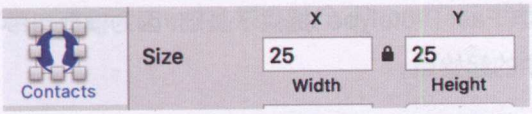
将编辑或返回按钮的左边距设置为 16，上下边距设置为 12。



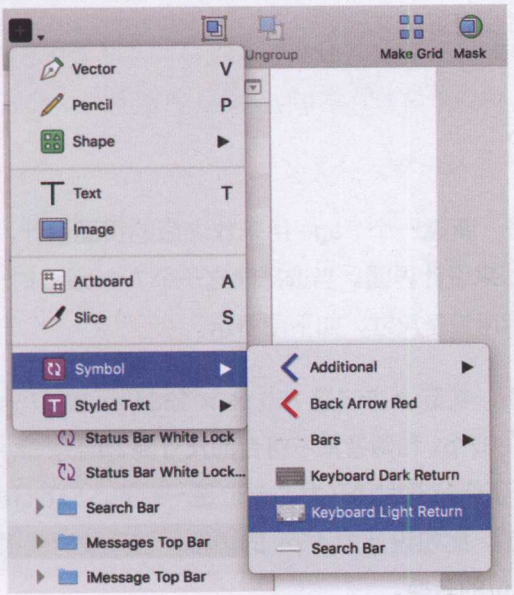
如何查看间距值？

选定元素后，按住键盘 Alt 键，移动鼠标到其他元素上，便能看到间距值。

Tab bar 设置为适合手指触摸大小的 25 × 25。



使用模板启动项目还可以方便地插入供项目重复使用的 Symbol，比如需要在界面中多次插入系统默认键盘，便可如此地操作。



什么是 Symbol 和 Styled Text？

比如返回按钮，需要在很多界面中用到，如果每次都复制粘贴，显得笨效率又低，所以可以把它定义成一个 Symbol，在使用的时候直接插入就可以了。

在顶部菜单栏选择 Create Symbol，右侧 Inspector 中为 Symbol 命名，然后在工具栏 Insert - Symbol 中插入。

Styled Text 也是相同道理，只不过它是基于文本的定义，不是图形的，但操作方式都一样。

在项目中应该灵活运用 Symbol 和 Styled Text，这样可以让所用到的设计元素成为一个系统，查找、使用和更新都十分方便。

模板为快速原型(Fast Prototype)提供了基础，因为模板自带了很多方便的组件，所以也是项目设计中的起始点。

由于模板能够帮助设计者快速建立起项目可能使用到的数据资源，就像每次做饭时食材和配料都已经准备好了一样，所以把它保存，在菜单栏中选择 File - Save as Template，命名后单击确定就可以保存备用了。在菜单栏选择 File - New from Template，选中刚才保存好的文档。

- Artboard -

按照之前的方法，新建一个 Page 用来放之后的页面设计，将它命名为 Oslo。接下来建立 Artboard 来设计界面。当选择建立一个 Artboard 时，会发现右侧有一些 Sketch 默认提供的常用设备尺寸，如下图所示。

以 iPhone 7 为例，实际上应该是 750 px × 1334 px 或 375 pt × 667 pt。为什么 Sketch 用 375 px × 667 px 和两者都不符合的尺寸呢？因为 Sketch 不知道你的显示器是什么样的，但为了保持原型和开发的单位是一致的，Sketch 让这里的 px 尺寸和真机的 pt 尺寸一样了。比如设计中 1 px 的边框，在开发时就很容易对应到 1 pt 的边框上，不用再做单位的转换。

▼iOS Devices	
iPad Pro Portrait	1024x1366px
iPad Pro Landscape	1366x1024px
iPad Portrait	768x1024px
iPad Landscape	1024x768px
iPhone 7 Plus	414x736px
iPhone 7	375x667px
iPhone SE	320x568px
Apple Watch 42mm	312x390px
Apple Watch 38mm	272x340px

选择 iOS Devices 中的 iPhone 7，建立一个 iPhone 7 尺寸的 Artboard，并改名为 Photos。

应该选用哪个尺寸的设备

遵循三个原则：

（1）现有设备。为了能在真机上更好地感觉自己的设计，要求设计尺寸和设备尺寸相符。在使用 Xcode 中的 Storyboard 进行原型模拟时，也会很有帮助。

（2）目标人群拥有的设备。比如应用的目标人群是中国人，那就应该在 5.5 英寸屏幕（iPhone 7/7S Plus）上进行设计，如果是欧美人，可以在 4.7 英寸屏幕（iPhone 7/7S）上设计，如果是非洲人，可以在 3.5 英寸屏幕（iPhone 4/4S）上……这样可以更好地保证你的设计能够服务好更多目标用户。

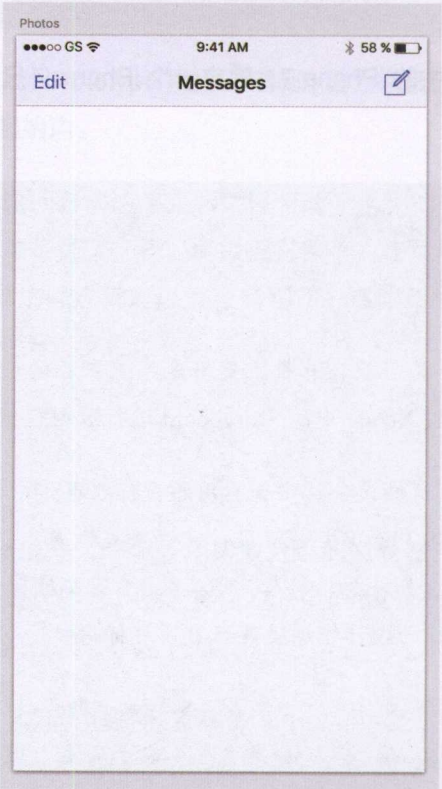
（3）保证设计的可伸缩性。如果在 3.5 英寸上做设计，可能很难想象在 5.5 英寸上的表现是什么样的。同样如果在 5.5 英寸屏幕上做设计，在 4 英寸屏幕上有些元素看起来可能会觉得有点别扭。所以最好是能够在一个大小适中的设备上进行设计。

- 导航栏 Navigation Bar -

选择 iOS 10 GUI Page 中的 Messages - TopBar，按键盘 Command+C[1] 进行复制，切换回 Oslo page，选中 Photos Artboard，按 Command+V 粘贴。

你看，不用去学习导航栏（Navigation Bar）的高度是多少合适，图标间距多少，一个基本导航栏就做好了！

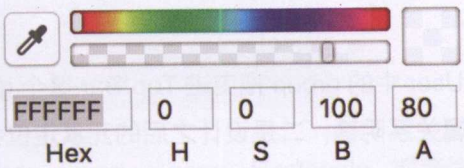
双击编辑，按 Delete 键删除。还记得怎么确认 Left Bar Button 与导航栏左侧的间距吗？按住 Alt 键，鼠标移动到导航栏就设计好了。这里应该是 16 的边距，确认后删除 Left Bar Button，这里我们用自己的图标。在 https://www.dropbox.com/s/bjqyk2cuge5paqv/icons.sketch?dl=0 下载 icons.sketch。



打开 icons.sketch，找到 User 图标，复制粘贴到 Photos Artboard 中，在

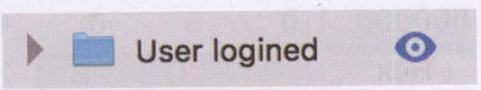
1 本书涉及的按键文字与图标对应关系为：Command(⌘)，Alt(⌥)，Shift(⇧)，Delete(⌫)。

Layers List 中选中所有 layer，在 Inspector - Fills 中，单击 Fill，将 Alpha（透明度）降到 80%，或者拖动上方透明度调节条。

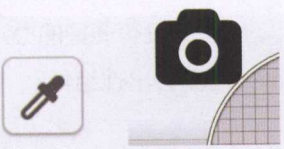


这是未登录状态下的图标。选中 User 这个文件夹，按 Command+D 复制新的文件夹并命名为 User logined，作为登录后的图标。用同样的方法将其中 layer 颜色改为 #FFD528。

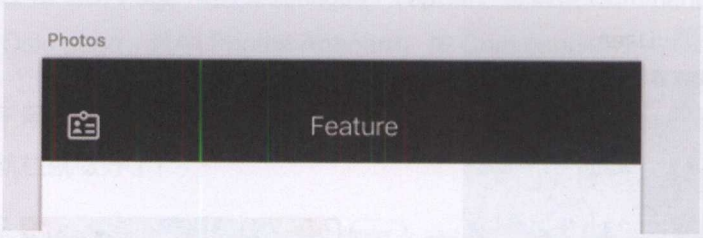
单击 User logined 文件夹右侧“眼睛”的图标来隐藏 layer。



在 Layers List 中选中 Background，然后在 Inspector - Fills 中单击 Fill。因为我们要做 Unsplash 的客户端，所以视觉风格可以保持一致——打开 unsplash.com，单击 Fill 中的吸管图标或按 Control+C，单击网站 Logo 吸取颜色，使导航栏颜色变为 #000000。



选中 title，使用同样的方法将颜色改为 #FFFFFF，Alpha 降到 80%。将内容改为 Featured，保持 Unsplash 默认的展示页面标题。



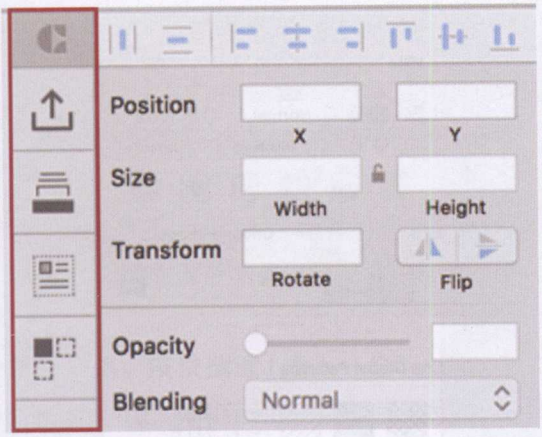
将 User 中的 group 拖曳进 Top Bar 这个 group 中。group 有两个好处：一是内容的从属关系明确，二是设计之后的元素定位会更清楚。这样应用的导航栏就设计成功了！

HSB/RGB

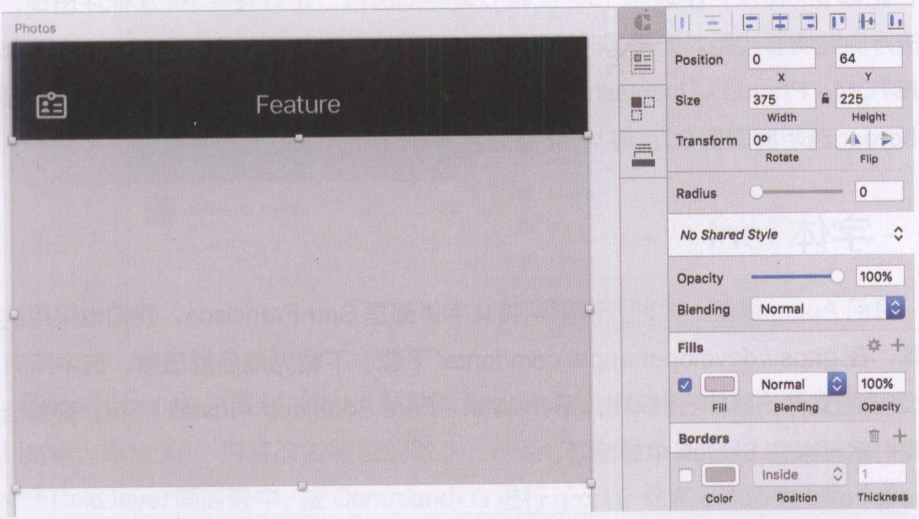
当鼠标单击图中 RGB 部分时，会看到 HSB 和 RGB 的模式切换。这其实是两种色彩的定义方案。这里不过多地解释这两者的区别，只不过有些颜色可能是 HSB，也可能是 RGB 形式，为了兼容 Sketch 提供了两种形式的切换。

- Craft -

在进行下面的设计之前，先介绍一款实用的 Sketch 插件——Craft（https://labs.invisionapp.com/craft）。因为在设计过程中，经常会碰到需要填充背景图或用户头像，或同一元素出现很多次等问题，但是又没有便捷的办法。Craft 就可以解决这些问题，所以下载安装好它，在 Inspector 左侧会出现组件。



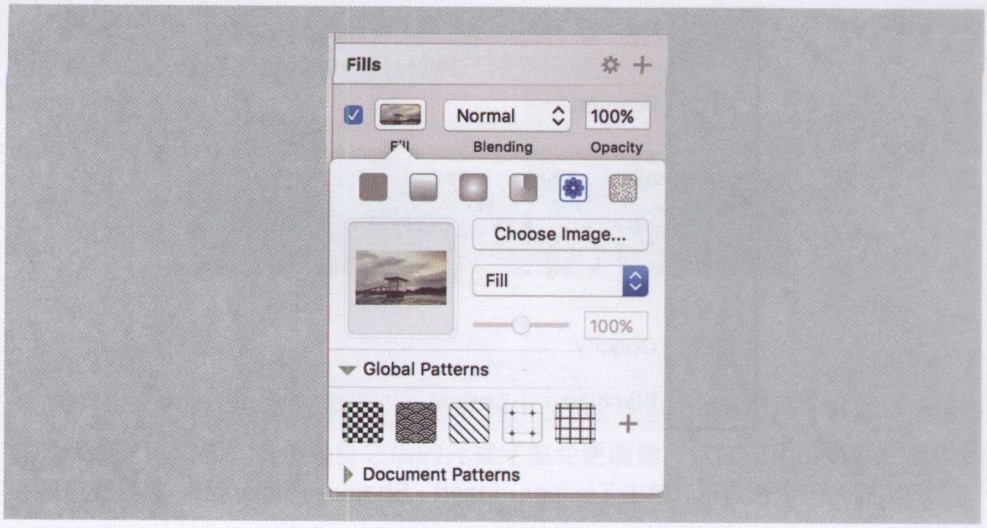
例如插入一个长方形，单击 Toolbar - Insert - Shape - Rectangle，或者快捷键 r，建议记下 Insert 中的各种快捷键。然后在导航栏下方画一个宽 375，高 225 的长方形，去掉 border，在 LayersList 将这个长方形改名为 photo(layer 的命名习惯首字母小写)。



选中 photo 这个 layer，在 Craft 中选择 Data - CUSTOM - Photos - Unsplash - Place Photos，这样来自 Unsplash 的图片就自动填充好了。

这是怎么做到的

其实这是利用了背景色填充里的图案填充，从 Unsplash 里抓来一张图案填充到这里。当然也可以选择自己喜欢的图案来填充。



在 Insert 中插入一个圆，或使用快捷键 o，按住 Shift 拖曳出一个正圆，宽高为 36，距离 photo12，左边距 16。在调节边距的时候，可以按住 Alt 让标注出现，同时按方向键调节边距。如果同时按住 Shift，可以以 10 个单位为跨度调节。在 Data - CUSTOM - Photos - Unsplash 中选择 People 分类、填充，将默认的 #979797 颜色的 border 透明度调到 80。将 layer 重命名为 avatar。

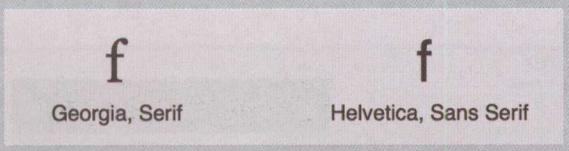
- 字体 font -

目前 Apple 设备上大部分的默认英文字体都是 San Francisco，我们也采用这种字体，在 https://developer.apple.com/fonts/ 下载。下载完成后解压缩，选中所有以 otf 结尾的文件，鼠标右键单击 Open With - Font Book.app - Install Font。安装结束后我们就可以在 Sketch 中使用了。

SF UI Display 和 SF UI Text 两种字体用法有什么区别？

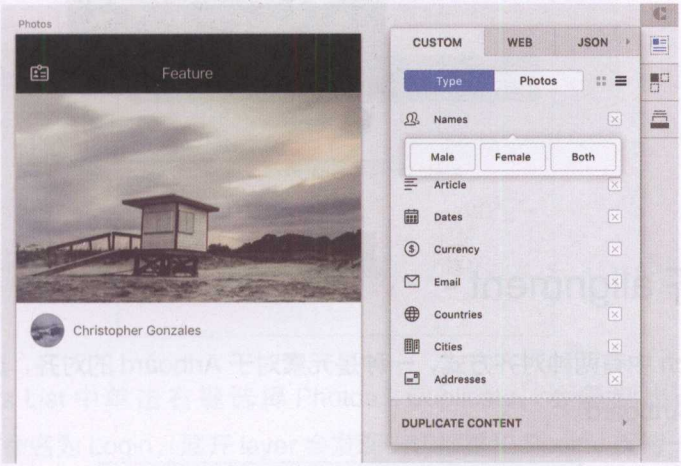
按照 Apple 官方的字体用法，当字号大于 20pt 时使用 SF UI Display，小于 20pt 时使用 SF UI Text，我们也遵从这个原则。

在英文字体中，字体类别分为两大类，一类是 Serif；另一类是 Sans Serif。Serif 在法语中意思是衬线，我们可以简单地理解为字母结尾处的修饰；Sans 意思是没有，所以 Sans Serif 可以理解为结尾处没修饰。



可以看出Apple全平台使用San Francisco这种字体，属于Sans Serif字体类别。目的是更容易让用户第一时间看懂屏幕上的内容，以及每个字母的表达意思。

接下来随机生成用户名。按快捷键 t 插入一个 text layer。Typeface 为 SF UI Text，Size 14，距离 avatar 10，并和 avatar 垂直居中对齐，在 Craft 中选择 Data - CUSTOM - Type - Names - Both（本书没有性别歧视）。

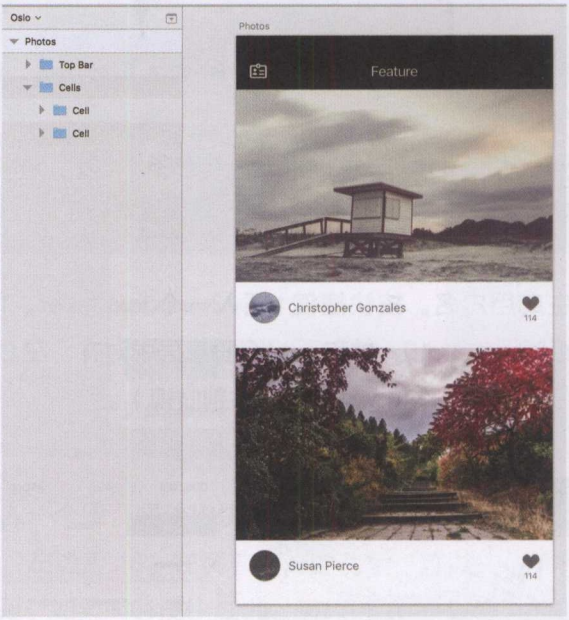


在 icons.sketch 中复制 heart，粘贴到 Sketch 中，右边距 16。在下方插入一个 text layer，Size 10，内容改为随意的数字，heart 上边距为 2。在 Layers List 中将 heart 和 text layer 同时选中，按 Command+G 进行 group，命名 group 为 Like（group 命名习惯首字母大写）。将 Like 和 avatar 垂直居中。将导航栏的部分全部整合，命名为 Cell。

选中 Cell，在 Craft 中选择 Duplicate - Vertical、Item count 2、Gutter 30。之后发现 Craft 很智能地生成了样式一样 Cell。将 Layers List 中的 Cell 全部选中 group，命名为 Cells。

这样应用的第一个界面就完成啦。注意你的界面可能和我的界面不一样，因为

我们的图片来源可能不是同一张。



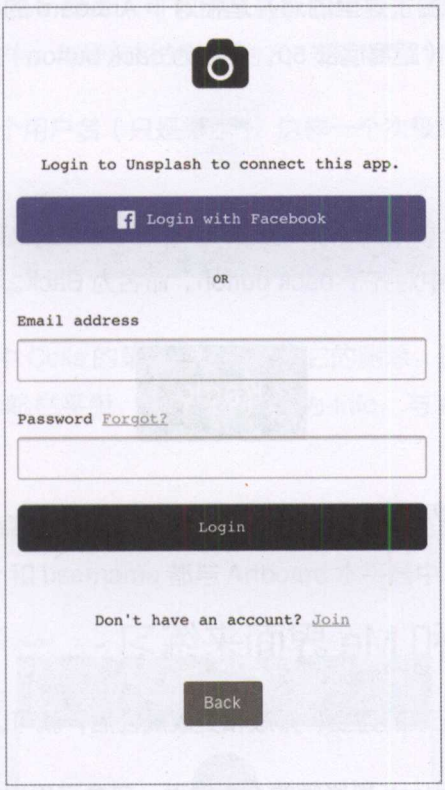
- 对齐 alignment -

在 Sketch 中有两种对齐方式，一种是元素对于 Artboard 的对齐，这种对齐方式的参照物是 Artboard。

还有一种对齐方式是文字的对齐，对文字的对齐仅限于文字使用，其对齐参照是自身。

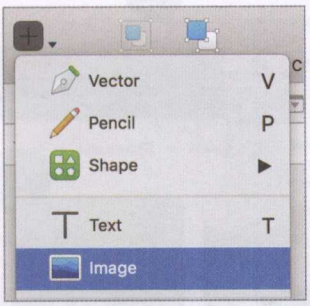
接下来完成登录界面。现在大部分的应用都具有注册登录和匿名两种状态，一般情况下注册登录后会获得更精准的内容推荐、跨平台的数据同步和个人资料的收集归纳。在 Unsplash 中登录也是类似的作用，Unsplash 通过内嵌的网页，用户登录后授权应用可以使用的权限后完成登录。因此这个界面的设计十分简单，只需一

个内嵌的网页和返回到上一级界面的按钮就可以了。设计完成后的界面如下图所示。



在 Layers List 中单击右键选择 Photos - Duplicate，会复制出一个相同的 Artboard，重命名为 Login，展开 layer 会发现一切都是和 Photos 保持一致的。

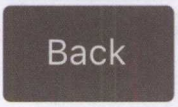
首先在手机上打开 unsplash.com，截取一张登录的页面，回到 Sketch 中选择 Insert - Image。



然后插入刚才的截图，命名为 login screenshot。

再按 U 键插入一个圆角的矩形，设宽 68，高 38，背景颜色为 #000000，60% 的透明度，居中对齐，由于这里的对齐是相对于 Artboard 的，所以直接点击下面这个图标来实现水平对齐（距离底部 50，命名为 back button）。

再按 T 键插入一个 text layer，内容为 Back，14 号字，颜色为 #FFFFFF，80% 的透明度，上下左右居中对齐于 back button。命名为 Back。



这样就完成了登录界面。

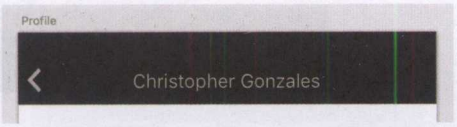
- 用 Profile 和 Me 界面来练习 -

在了解了 Sketch 元素的建立、排版和更改属性后，就可以进行练习了。

Profile 界面是 Unsplash 摄影师的个人主页，包含了用户的基本信息和发布过的照片。在设计完成后应该是下图所示的样子。

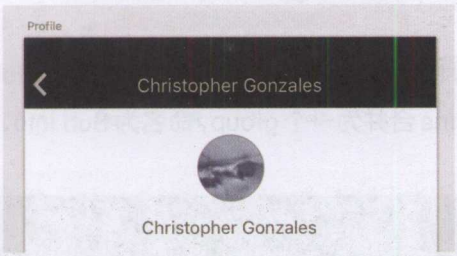
复制 Photos Artboard，重命名为 Profile。删除导航栏中的 User。从 iOS 10 GUI 中复制 Back Arrow Blue（可以用 Layer List 下方的 Filter 来搜索），回到 Oslopage 粘贴到导航栏中，与导航栏左边距为 8，下边距为 11，颜色为 #CDCDCD。

将 Feature 改为一个用户名（只是举例），这样一个次级的导航栏就做好了。

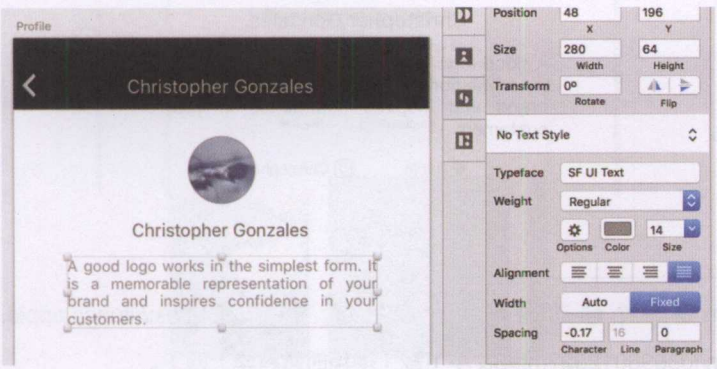


只保留 Layer List 中 Cells 的第一个 Cell，其它的删除。选中用户名的 text layer 和 avatar，拖曳到与导航栏平级，group 重命名为 Info，与 Artboard 水平居中，然后删除 Cells。

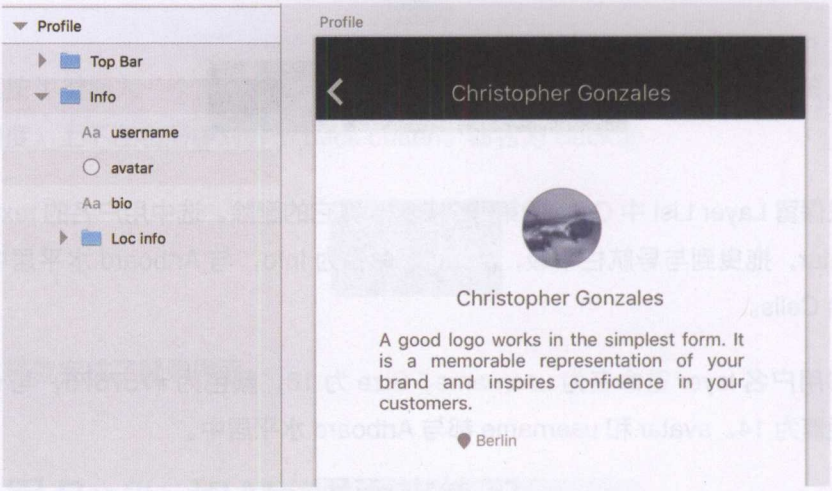
将用户名 layer 重命名为 username，Size 为 16，颜色为 #757575，与 avatar 垂直距离为 14。avatar 和 username 都与 Artboard 水平居中。



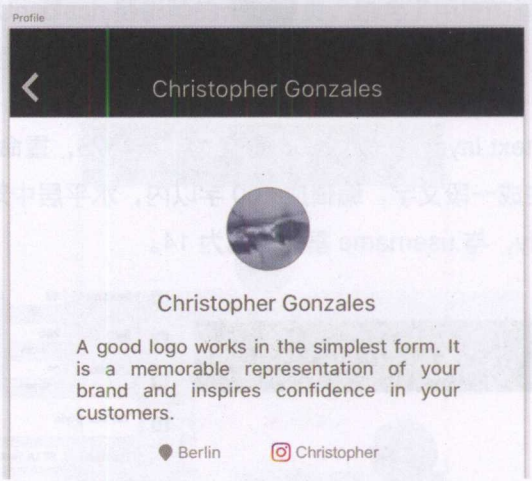
在 Info 中插入 text layer，宽为 280，颜色为 #757575，重命名 layer 为 bio。选中后用 Craft 随机生成一段文字，编辑成 100 字以内，水平居中对齐，Alignment 设置为最右侧的 Justify，与 username 垂直距离为 14。



从 icons.sketch 中复制 location 图标，粘贴在 Info group 中，颜色为 #9D9D9D。插入一个 text layer，随意命名为一个地点，layer 命名为 loc，Size 为 12，颜色为 #9D9D9D，loc 与 location 的水平距离为 4。将 loc 和 location 合并为一个 group 命名为 Loc info，属于 Info 内。和 bio 的垂直距离为 14。



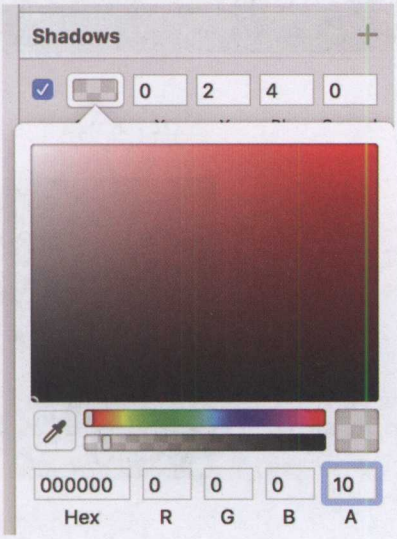
用同样的方法添加 Instagram[1] 信息，group 命名为 Portfolio info，与 Loc info 间距为 36。将 Loc info 和 Ins 合并为一个 group，命名为 Sub info，与 Artboard 水平居中。



1 Instagram 是一个移动端照片分享社区，可以在 https://itunes.apple.com/us/app/instagram/id389801252?mt=8& 中下载（使用时需科学上网）。

由于有些用户没有 Instagram 信息，但有自己的个人网站，所以这种情况也需要考虑到设计中。在 icons.sketch 文件中复制 portfolio 图标到 Sub info 文件夹中，位置和 Instagram 图标保持一致，并隐藏该 layer。

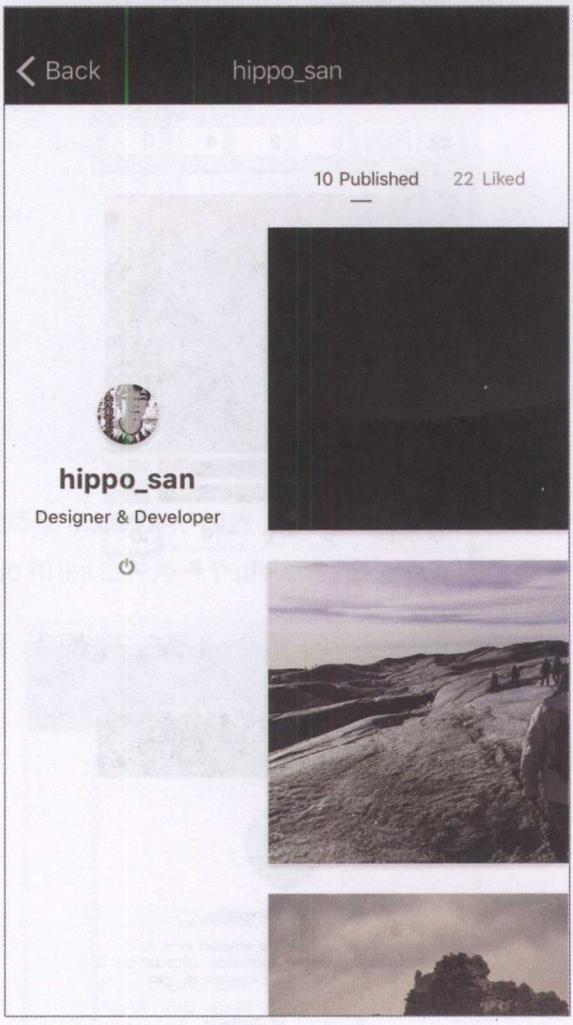
插入一个圆角为 3，宽为 180 的正方形，Position X 为 -42，将 Shadows 透明度改为 10，其他保持不变，如下图所示。



用 Craft 水平复制两个 Gutter 为 20。用 Craft 填充三张图片，如下图所示。

将三个正方形合并为一个 group 命名为 Photos，与 Info 垂直距离为 44。同时选中 Info 和 Photos，与 Artboard 垂直居中。这样 Profile 页面就做好了。

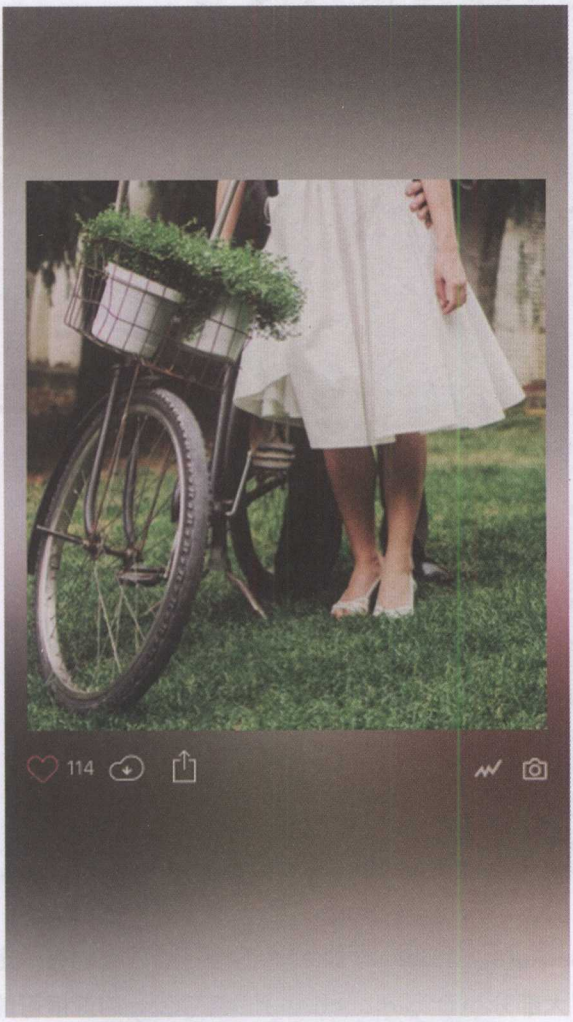
Me 界面是登录后的个人页面，里面记录了个人发布和添加的喜欢图片。这个界面就由你自己来完成吧，完成后如下图所示。



- 同一界面的多种不同状态 -

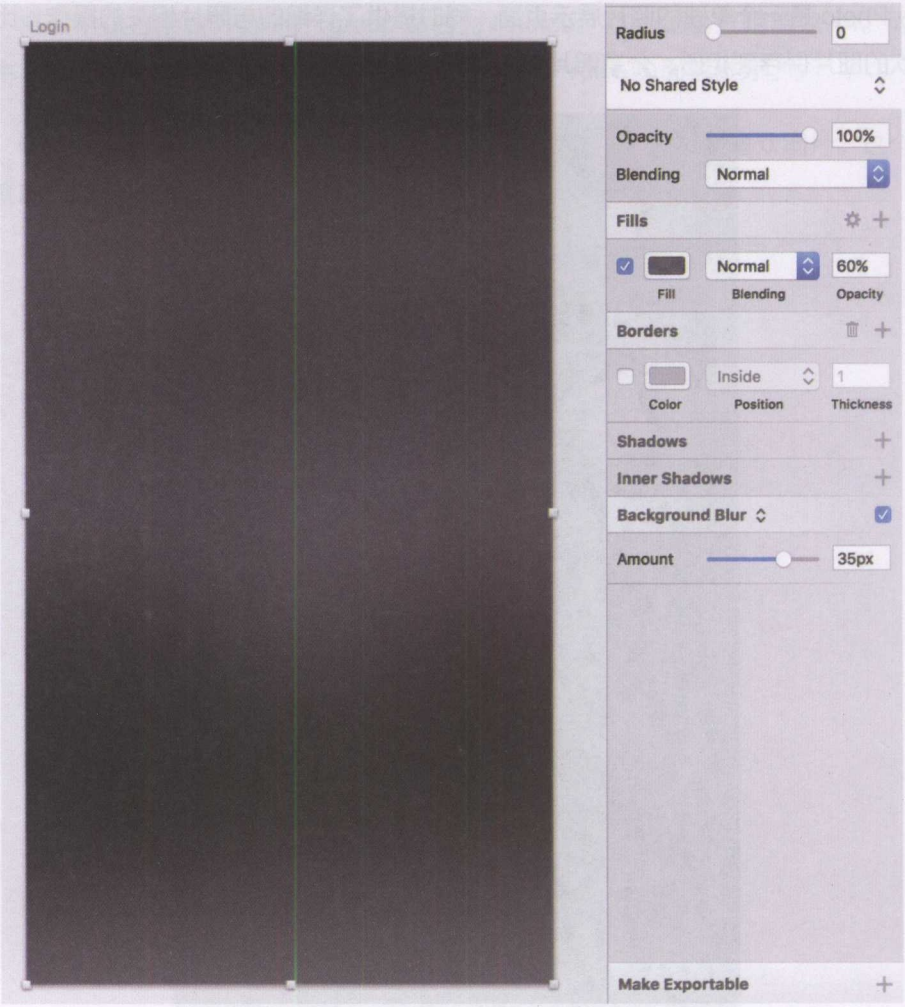
同一个界面往往会出现多种不同状态，比如单击一些按钮后会出现弹窗、提示、按钮状态的变化……在设计时可以针对这些不同的状态单独建立 Artboard。

Photo 是一个单独的照片展示页面，同时提供了详细的与照片相关的操作，比如喜欢的图片保存到相册、分享照片等。设计结果如下图所示。



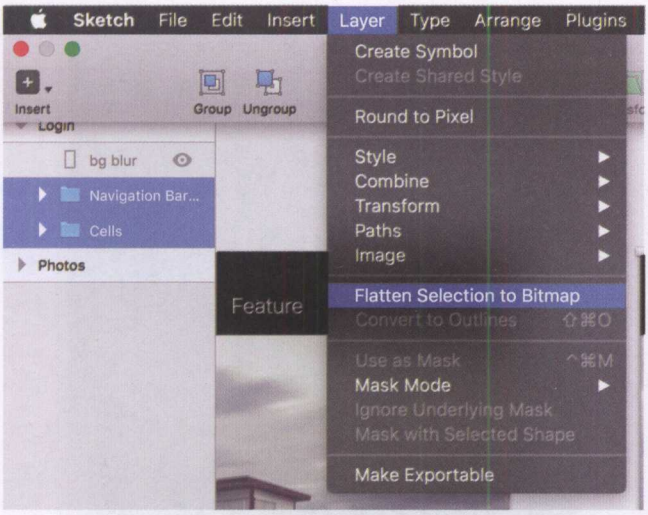
可以看到这个页面有很多按钮，但都不会改变页面本身的内容展示。下面就来一步一步设计这种情况的页面吧。

首先对背景进行模糊处理。复制 PhotosArtboard，重命名为 Photo。按住 r 键画一个覆盖整个 Artboard 的长方形，去掉 border。Color 为 #000000，Alpha 为 60，重命名 layer 为 bg blur。单击 Inspector 最后一项 Gaussian Blur，选中 Background Blur，Amount 设置为 35 px。

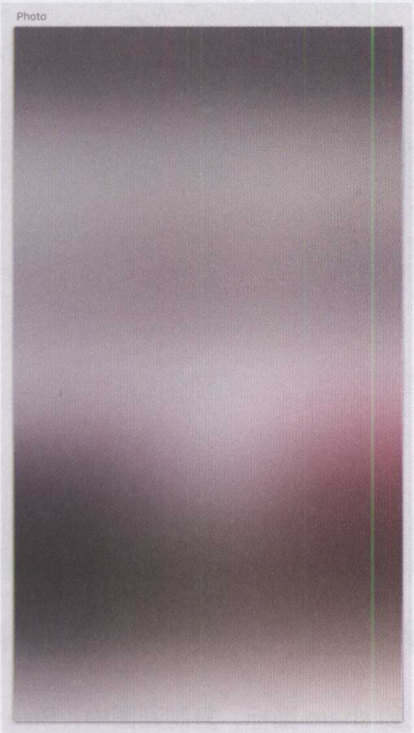


注意，使用 Background Blur 对 Sketch 的运行处理性能有一定的内存占用，如果在设计中使用了大部分的 Background Blur，继续设计时速度可能会变得很慢。因为 Sketch 在这个时候要对 Blur layer 后面的每一个图形进行处理，需要运用大量的运算，如果要提升效率，只要将后面的这些图形变成一个图形就可以了。

首先让 bg blurlayer 隐藏。然后选中除 bg blurlayer 以外的全部内容，单击顶部菜单栏中的 Layer - Flatten Selection to Bitmap，这样背景就变成了一张静态图片。之后再进行 Background Blur 操作时，Sketch 只是对这张图片进行处理，这样效率会提升很多。



将 Bitmap 重命名为 bg，bg blur layer 恢复成可见。



画一个长方形，设宽 345，高 363，与 Artboard 顶部边距为 115，去掉 border，填充 Unsplash 图片，命名为 photo。

将 icons.sketch 中的 heart-outline 复制到 photo 中，与 Artboard 左边距为 15，与 photo 下边距为 12。

创建一个 text layer，内容为随意的数字，用来表示红心数，Size 为 12，颜色为 #FFFFFF，Alpha 为 70，与 heart-outline 垂直居中，重命名 layer 为 heart count。

将 download icon 复制到当前，与 heart count 垂直居中，左边距为 10。

在 iOS 10 GUI 中找到 share 图标，复制到图中，Size 调整为宽 15，高 22，与 download 右边距为 20。

将 layer group 两个图标重命名为 Ops，与 photo 下边距为 12。

采用同样的方法，将 camera 和 statistics 复制过来，与 Ops 垂直居中，两个图标的间距为 15，将 group 重命名为 Info。

由于每个按钮都有着不同的状态和操作反馈，所以在设计阶段，这些都要逐一设计。

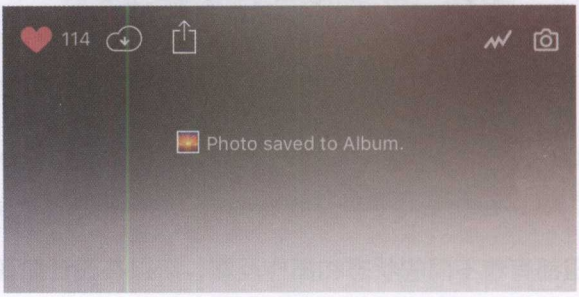
单击红心

接下来完成红心单击状态的设计。Duplicate Photo，将 Artboard 重命名为

Photo-ops。将 heart-outline 的 Borders 去掉，在 Fills 中填入 Borders 的颜色 #D96969。

单击下载

在 Photo-ops 中创建一个 text layer，内容为 Photo saved to Album，12 号字，颜色为 #FFFFFF，透明度为 70%。layer 重命名为 tip，不属于任何 group，与 Artboard 水平居中，距离 Artboard 下边距 88。



单击分享

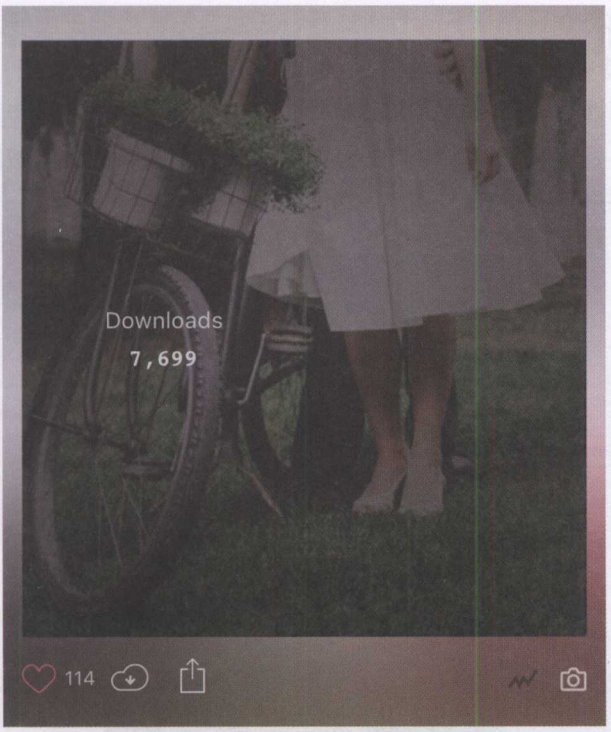
由于是使用 iOS 系统的分享，有系统自带的样式，所以这里就不需要进行特殊设计了。

单击统计

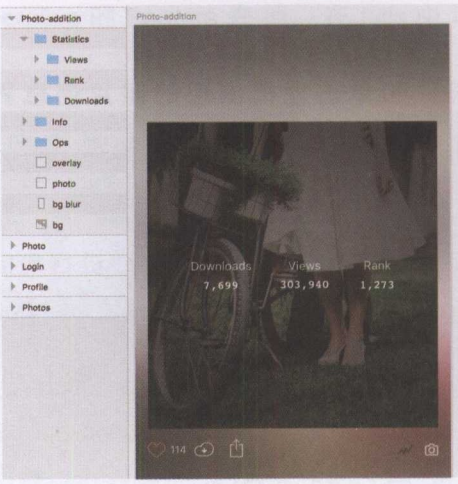
接着完成单击统计和单击相机出现的照片信息界面。Duplicate Photo Artboard，重命名为 Photo-addition。将 statistics 颜色改为 #000000，alpha 为 40，用来表示统计状态。

按住 Alt，在 layer list 中向上拖曳 photo layer 进行复制，保持新的 layer 在 photo 上面，重命名为 overlay。在 Fills 中将颜色设置为 #000000，alpha 为 70。

建立两个 text layer，一个图层为 Downloads，alpha 为 80，另一个图层随意填写一个数字，alpha 为 90，数字字体为 CourierBold，Size 均为 14。数字和 Downloads 垂直间距为 6。将它们组合命名为 Downloads，与 Artboard 左边距为 37。

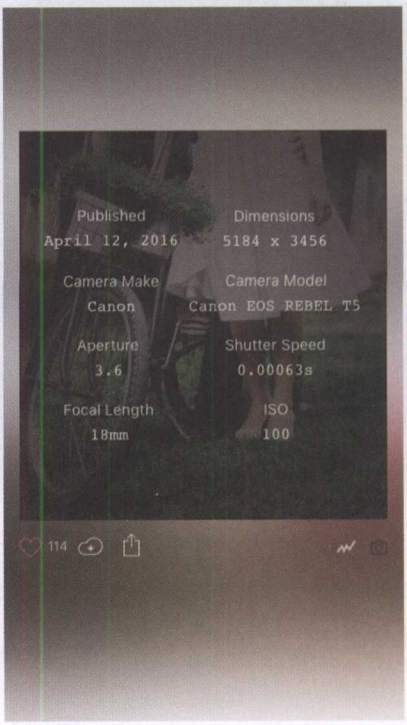


用 Craft 横向复制两个 Downloads，Gutter 为 30，合并为一个 group 后命名为 Views 和 Rank，再将 Downloads、Views 和 Rank 三个 group 进行 group 后命名为 Statistics，与 Artboard 水平居中，与 overlay 垂直居中。



单击相机

参考单击统计的做法，对不同的信息部分 group 后重命名 group 为对应的标题，最外层的 group 命名为 Exif。将单击相机后显示摄影信息的界面做出来，完成后的设计图如下图所示。



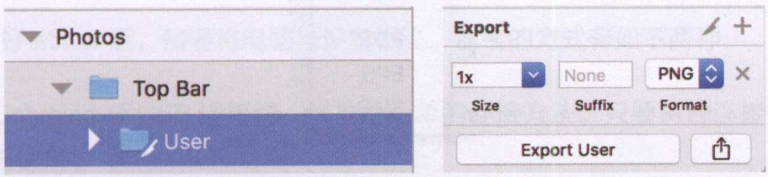
在所有这些结束后，应用的界面设计就完成了。完成后的 Sketch 文件可以在 https://www.dropbox.com/s/h65c23esv8qb6sv/Oslo.sketch.cpgz?dl=0 下载查看。

在整个设计过程中，相信你已经学会了使用 Sketch 最主要的功能和结构来实现自己的设计，当然这些设计也完全能直接应用在之后的开发上。但是怎么为开发提供设计资源，怎么在手机上“真实”地看到自己的设计图，怎么管理设计资源和 Sketch 的其他小技巧，这些都还不是很了解。接下来会围绕着这些问题逐步讲解，这些问题也是专业设计师管理自己的作品和协作的技能。

- 导出 export -

设计资源的导出主要是为了之后进行开发设计，几乎所有的视图元素都可以在 Sketch 内导出，无论是一个形状，还是 group，或者 Artboard，都可以导出（但 Page 暂时还无法导出）。

导出的方式也很简单，例如选择 User 这个图标，虽然它是一个 group，但可以作为一个完整的图标导出。选中 User 这个 group 后，在右下方选择 Make Exportable，这时能看到 User 的文件夹图标上多了一把小刀图形，说明这个是“可被导出的”。



Export 的一些选项如下。

- Size：对于 Universal App[1]，需要三种尺寸的设计资源 1x、2x 和 3x。
  - ◇ 1x：针对非 Retina 屏幕的 iOS 设备。
  - ◇ 2x：针对 Retina 屏幕但非 iPhone6、iPhone 6 Plus、iPhone 7、iPhone 7 Plus 的设备。

1 Universal App 即指同一个 App 可以在所有 iOS 设备上运行，比如 iPhone 和 iPad。

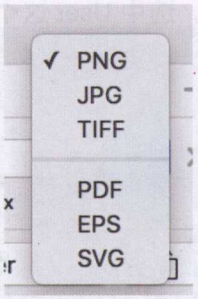
◇ 3x：针对 iPhone 6、iPhone 6 Plus、iPhone 7、iPhone7 Plus 的设备。

而导出的尺寸应该基于设计本身。如果 Artboard 是按实际 px 来设计的，比如 iPhone 7 的实际分辨率为 750px × 1334px，这个时候如果按照 1x 导出，那么实际上在之后的开发中会把导出的资源作为 2x 来使用，因为这个资源本身就是实际 Retina 像素的分辨率。如果要想获得开发时用于 1x 的设计资源，就需要在 Sketch 中按 0.5x 来导出。

但对于 Oslo 这个例子，按照 pt 的单位，只要按 1x、2x、3x 导出就行，对应到后期的开发也是对应的 1x、2x、3x。这也是为什么最初要求用 Sketch 默认的分辨率来进行设计的原因。

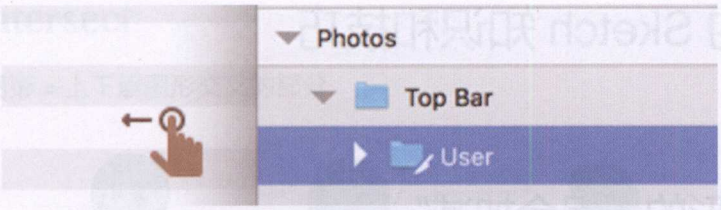
- Suffix：可以定义导出的资源在 Sketch 中元素的命名后面附加什么名字，一般用于尺寸说明上，比如在 Suffix 中填写 @2x，导出的资源名称即为 User@2x。

- Format：展开后可以看到选项被分为了两组。



横线以上部分为位图（Bitmap），以下部分为矢量图（Vector）。一般情况下位图以 PNG 格式导出，好处是支持透明度，色彩空间也更广，代价是文件内存相对大一些。矢量图对于应用开发应该选择 .pdf 格式。Web 开发应该使用 SVG，在多尺寸设备都能够清晰显示的基础上，SVG 格式还方便于动画制作，不过这里就不详细说明了。

如果导出为 PNG 格式的图片，还有一个快捷的方法，就是在 Layers List 中选中要导出的资源，然后直接拖入需要保存的文件中。



当然如果已经导出，而且设置好了相应的格式，那么这种拖曳的方式也是直接导出为相应的格式。

因为 PDF 格式的图标更加灵活，所以请将项目中的图标导出为 PDF 格式供之后使用。

- 在“真机”上看到自己的设计 Mirror -

因为不同的屏幕设备，其色彩区域、显示精度不同，导致最终在实际设备上显示的效果会有所差异，所以在最终设计定稿时都需要在实际设备上试一下实际效果。

Sketch 提供了 Mirror，可以将 Sketch 中的 Artboard“投射”到连接的 iOS 设备上。设备端需要在 App Store 中免费下载 Mirror，也可以在这里了解 https://www.sketchapp.com/features/#mirror。

下载好 Mirror 后，需要将电脑与设备相连。连接的方式有如下两种。

- Lightning 至 USB 连接线：这是最简单的连接方法。只要将线连接电脑和设备即可。
- 使用 WiFi：确保电脑和设备处于同一 WiFi 下。

打开设备上的 Mirror App 后，单击电脑 Sketch Toolbar 上的 Mirror，于是手机就能看到电脑上的 Artboard 了。可以左右滑动来切换，非常方便。

注意，因为我们一直是用 pt 的单位做“模拟”，所以在真机上的显示并不是 Retina 的。

有用的 Sketch 知识和技巧

- 图形的“组合加减” -

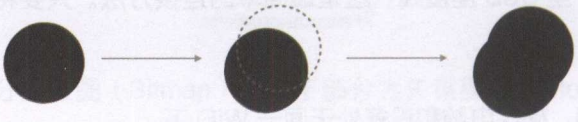
在 Sketch 中提供了多个图形的操作，以及多个图形的组合操作，这样可以更有效率，也更能准确地设计出想要的图形。比如 iOS 控制中心的“请勿打扰”功能的图标。

例如画中的月亮虽然可以用 Vector 一笔一笔地画，但这样做既不准确也浪费精力。显然有简便的方法，比如直接利用 Sketch 中的图形减法。

下面是 Sketch 中提供的图形组合操作方式，下面以图形来举例。

1. Union

最终图形 = 下层图形 + 上层图形

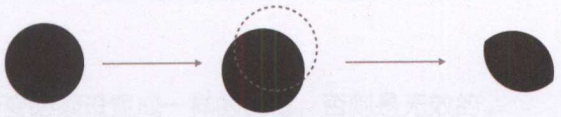


2. Subtract

最终图形 = 下层图形 – 上下层图形交叉的部分

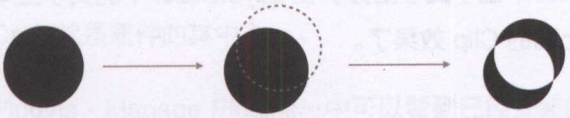
3. Intersect

最终图形 = 上下层图形交叉的部分



4. Difference

最终图形 = 下层图形 + 上层图形 - 上下层图形交叉的部分



- 文字操作 -

在 iOS 的界面中有时会遇到特殊效果的文字，下面列举了几个常用的实现方式。

Transform

可以看到上图中未被选中的字体都在一定程度上发生了形变。

其实只要将文字转化成 Outline 就可以了。在需要变形的文字上单击鼠标右键，选择最后一项 Convert to Outlines。可以看到 Layers List 中文字的字母已经全部转化成了 Path，这样我们就可以设计变形了。

选择 Toolbar 上的 Transform，选中其中一个锚点后拖曳成自己想要的效果。

Clip

同样将文字 Convert to Outlines 后，插入一张图片，将图片和文字 group，注意图片在 Layers List 中位于文字上方。在 Layers List 中的文字上单击鼠标右键选择 Mask，就能出现上面的 Clip 效果了。

Gradient

同样将文字 Convert to Outlines 后，在 Inspector-Fills 中选择想要的渐变。

- Alpha Mask -

有时会看到一些音乐应用的唱片封面有类似下面底部渐变的效果，如下图所示。

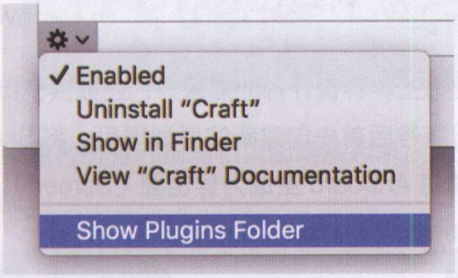
画两个大小和位置完全一样的图形，将上层图形填充一张图片，下层图形单击鼠标右键设置为 Mask，选中菜单栏的 Layer - Mask Mode - Alpha Mask 。将下层的渐变选择为 Linear Gradient，两端颜色设置为完全一样的色值，调节不同的 Alpha 就完成了。

注意这里的渐变必须设置成一样的色值，否则是无效的。

- 插件 plugin -

Sketch 之所以强大且有着更多的扩充可能性，是因为它友好地支持插件，之前我们一直使用的 Craft 就是插件的其中之一。

在菜单栏的 Plugins - Manage Plugins··· 中可以看到已经安装的插件。需要注意的是部分手动安装的插件可能不会显示在这个列表中，如果想查看这部分安装的插件，需要打开 Plugins 的安装文件夹，如需删除插件，直接删掉整个文件夹就好。



如果想安装新的插件，可以单击界面右下角的 Get Plugins 浏览发现。

用 inVision 来制作原型和管理设计（Bonus）

这部分内容和 Oslo 的制作没有太大关系，如果想直接继续 Oslo 的制作，可以跳过本节。

如果你是一位专职设计师，你一定有类似以下的需求：

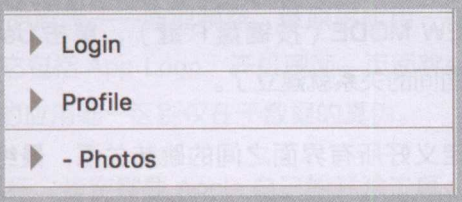
- 简单页面间跳转关系的原型呈现。
- 跟各种其他角色，比如产品经理、工程师或设计师之间的协作需求。
- 设计的版本管理。
- 设计的进度管理。

这些需求并没有在 Sketch 中得到完全的满足，能够实现类似需求的应用有很多，在这里介绍其中一款——inVision（下载地址是 https://www.invisionapp.com/）。

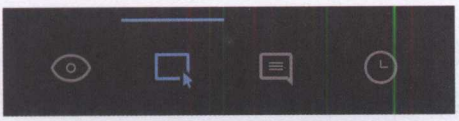
inVision 除了能满足以上种种设计师的需求外，还提供了良好的体验和周到的服务。以 Oslo 为例，注册账号后首先创建新的原型项目，将 Sketch 文件直接拖曳到页面中，inVision 会自动将 Artboard 生成为导出图（Screen），也可以通过鼠标拖曳来重新排序。同时 Sketch 文件会保存在 ASSETS 中。

删掉不想导出的资源

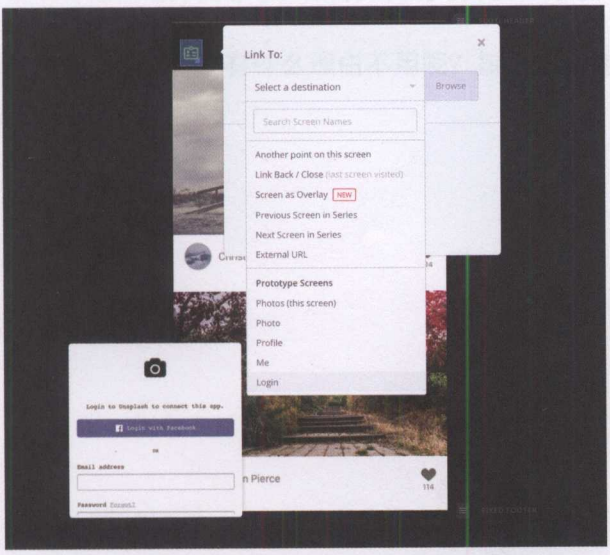
在 Oslo 中有些在原型制作中涉及不到的设计资源，可以不必导出。方法是在 Layers List 中找到该资源，在命名前加上“-”，这样在同步到 inVision 时就不会处理这类资源了。



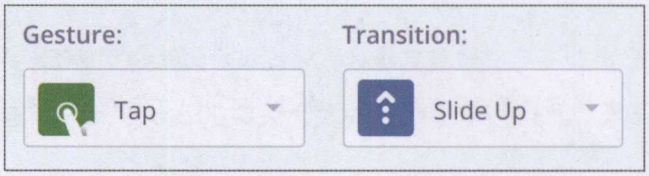
单击 Photos 界面后，会进入到原型制作界面。在此界面可以修改模拟器的外观、就某个设计部分添加设计评论。不过这里最核心的功能是为应用界面制作带跳转关系的原型。单击最下方的 BUILD MODE。



现在我们的任务就是将所有的按钮和页面连接起来。比如框选左上角的 User 图标，链接到 Login 界面。



Gesture 定义需要操作的手势，Transition 定义从当前界面如何过渡到下级界面：



切换回 PREVIEW MODE（按键盘 P 键），单击 User 图标，就进入了界面 Login，这样两个界面间的关系就建立了。

用同样的方法定义好所有界面之间的跳转关系，最终完成的原型可以参见：https://invis.io/699R1P28K#/211391744_Photo。

完成原型后，在 iOS 设备端下载 inVision Viewer，或者在这里下载：https://invis.io/699R1P28K#/211391745_Photos。

下载完成登录后即能在设备上真实体验自己做的设计了。

除了原型制作以外，inVision 还提供了界面的交叉评论、看板进度管理、实时协作、版本管理、多平台集成等以设计为中心的工具，不过由于已经超出了本书的范围，在这里就不详细介绍了。

至此，用 Sketch 进行的设计工作就结束了。笔者个人最大的感受是 Sketch 以“设计想法”本身为中心来提供服务。如像素对齐、样式 / 元素复用、不同尺寸的资源导出等以前使用其他设计软件需要操心的问题，Sketch 在最大程度上方便了设计者，而且最终设计成果对于后期的开发也完全可用。

设计工作结束了吗？答案是：还没有，那继续我们的 Oslo 制作旅程吧。对于设计来说，还有一个最重要的阶段——原型制作。像原生应用一样可以在不同的 iOS 设备上“真实”地运行，它包括 App Logo、开机画面、页面跳转关系和动画等，它与最终提交到 App Store 的应用唯一区别仅在于数据的真伪。

能够在 iOS 设备上运行，肯定需要 Apple 自己的开发工具，这就是 Xcode。很多设计者对 Xcode 望而却步，但 Xcode 在实现设计操作和逻辑方面非常直观。另外用 Xcode 来进行原型制作的好处如下。

- 程序开发时可以直接在 Xcode 制作出的原型基础上来做，省去了大量的重复劳动。
- 可以真正运行在设备端，不再有“假图”的感觉。
- 可以兼容所有 iOS 的设备分辨率，保证在任何设备中达到的设计目的都一样，也是对设计者的基本要求。
- 可以更了解 iOS 应用制作的思想，自己开发或与开发者沟通时更加顺畅。

使用 Xcode 有这么多优点，还有什么理由不用呢？接下来，进入本书的新篇章——用 Xcode 来制作原型。

原型

用 Xcode 制作可以在所有 iOS 设备上运行的设计原型。

二

Xcode 介绍

Xcode 是 Apple 官方唯一的 iOS App 开发工具。可以在 Mac App Store 中搜索下载，也可以在 https://developer.apple.com/xcode/download/ 下载。

Xcode 时常会提供稳定版和 beta 版两种版本，正式版所有人都可以下载安装，也可以提交应用到 App Store；beta 版只提供给加入了 Apple Developer Program 的人，它包含最新的 Xcode 的功能和 Swift[1] 的改进，并且往往会附赠一些 bugs，但是用 beta 版制作的应用无法发布到 App Store。可以在 https://developer.apple.com/videos/play/wwdc2016/413/ 了解到 Xcode 的一些基础知识。

我需要加入 Apple Developer Program 吗？

Apple 在 https://developer.apple.com/support/compare-memberships/cn/ 详细说明了这个问题。简单来说有以下需要时才加入 Apple Developer Program。

- 需将应用提交到 App Store 。
- 使用 beta 版的开发工具或者系统。
- 使用多样的功能，比如 In-App Purchase 、Universal Links 等。

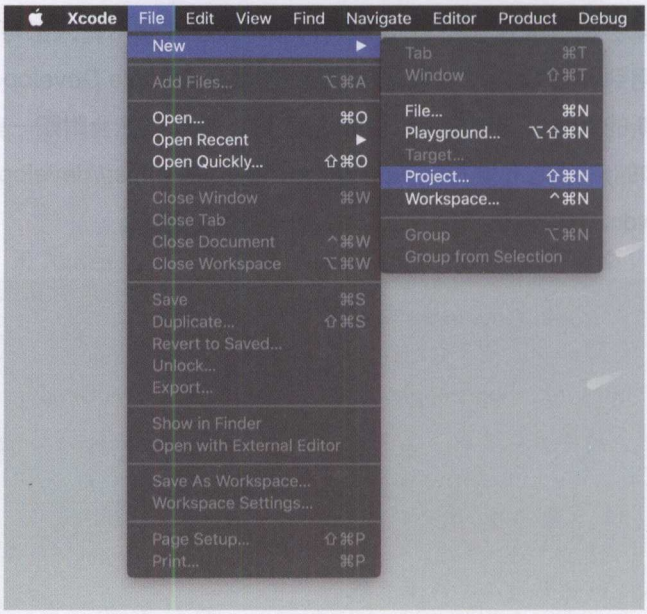
也可以在 https://developer.apple.com/programs/whats-included/cn/ 查看具体的权益。

加入 Apple Developer Program 一年的费用为 99 美元。

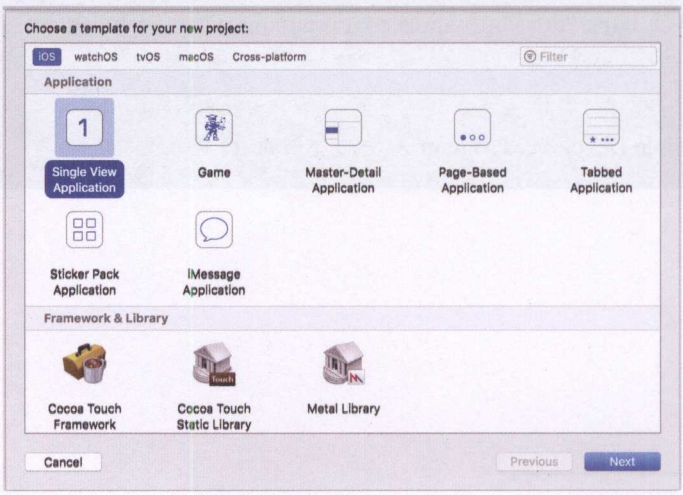
1 Apple 出品的编程语言，本书后面会介绍。

第一次 Build

打开 Xcode，在菜单栏中选择 File - New - Project，或者按快捷键 Shift+Command+N。

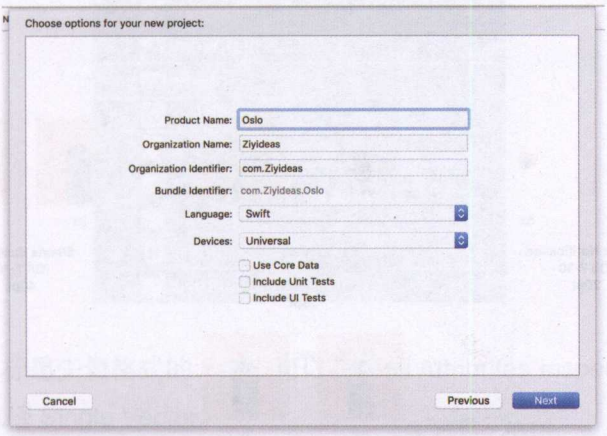


选择 iOS - Single View Application，单击右下角的 Next 按钮。

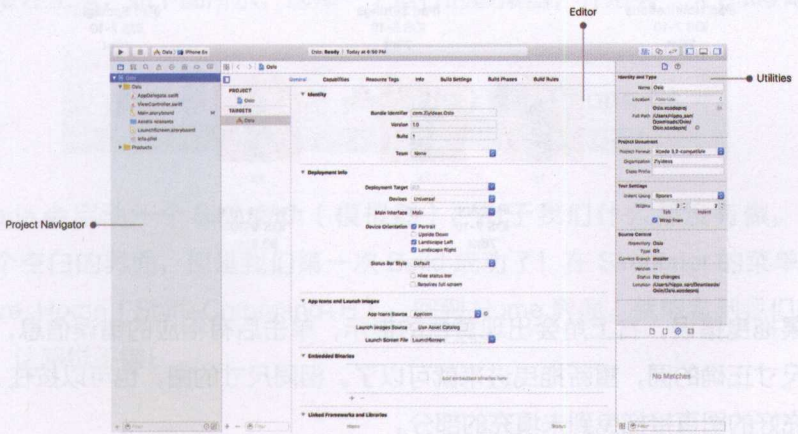


其他的几个选项都在一定程度上为我们定制了项目的结构，因为是学习实践，所以从完全空白的项目开始比较易于理解，因此选择 Single View Application。

Product Name 为项目名称，这里填写 Oslo。Organization Name 为组织名称，可以根据自己的需要填写。Organization Identifier 为组织识别 ID，也可以根据自己的需要填写。Language 为程序语言，这里选择默认的 Swift。Devices 为应用设备，因为 Oslo 在所有 iOS 设备上都可以运行，所以选择默认的 Universal。单击右下角的 Next 按钮。



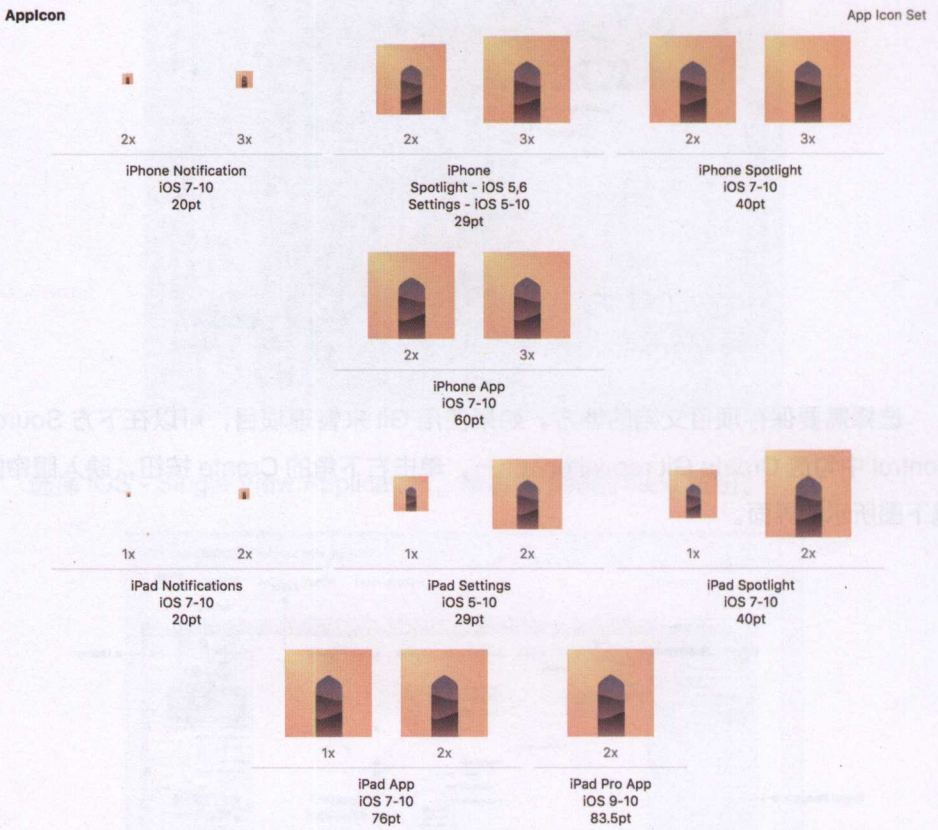
选择需要保存项目文档的地方。如果使用 Git 来管理项目，可以在下方 Source Control 中勾选 Create Git repository on…，单击右下角的 Create 按钮，映入眼帘的是下图所示的界面。



找到 Project Navigator，单击 Assets.xcassets，会发现有一个默认存在的 AppIcon，这是应用的 Logo，在这里设置好后，就能在 iOS 设备上看到 Oslo 的图标了，是不是很酷?

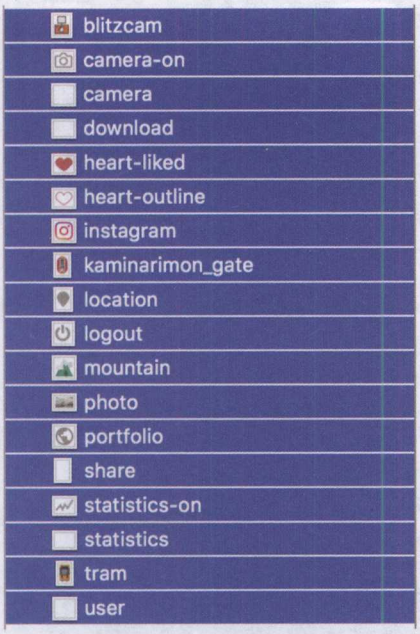
在 https://www.dropbox.com/s/kzkk9bz61qyiwis/Oslo%20-%20App%20icon.zip?dl=0 下载 AppIcon，解压缩。此 App Icon 用 Prepo 生成，可以在这里下载安装 https://itunes.apple.com/us/app/prepo/id476533227?mt=12。

回到 Assets.xcassets - AppIcon 中，将 App Icon 文件中对应尺寸的图拖曳进相应的格子里。

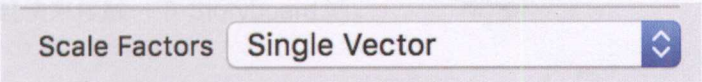


如果拖曳错误，右上角会出现黄色的提示，单击后有相应的错误信息，这时只要找到尺寸正确的图，重新拖曳进来就可以了。相同尺寸的图，也可以按住 Alt 键将已经填充好的图直接拖曳到未填充的部分。

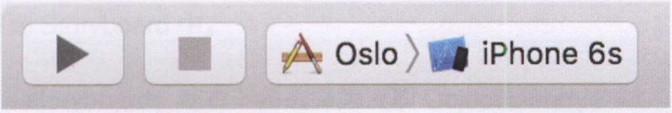
将之前导出的所有应用的图标拖曳进来。



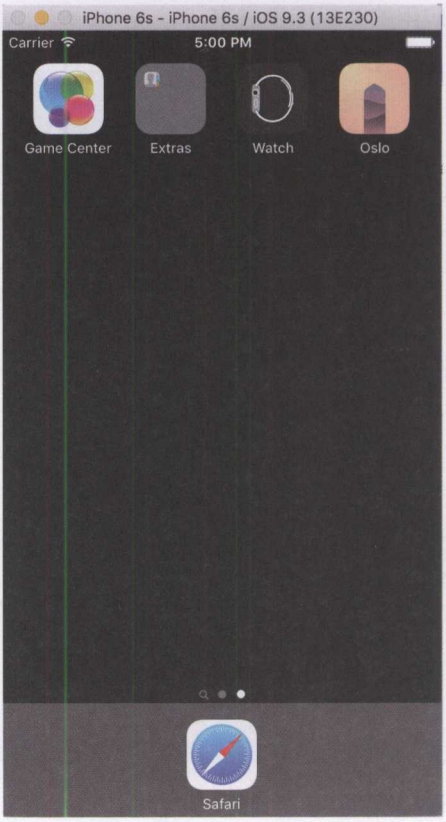
因为这些图标是矢量格式的，所以可以在右侧 Attributes inspector - Devices - Scale Factors 选择 Single Vector。



在菜单栏选择 Product – Run(Command+R，记住这个快捷键，以后会经常用到)，在 Xcode 左上角，如下图所示，选择一个运行的模拟器，开始第一次 Build 吧！



Xcode 会启动一个 Simulator（模拟器），由于我们什么都没有做，所以只出现一个空白的界面，但是我们第一次 Build 成功了！在 Simulator 的菜单栏选择 Hardware–Home（Shift+Command+H），回到 Home 界面，就能看到我们的 Oslo 应用了，这感觉不错！


iPhone 6s - iPhone 6s / iOS 9.3 (13E230)
Carrier
5:00 PM
Game Center
Extras
Watch
Oslo
Safari

Storyboard

在 Project Navigator 中可以看到有两个以 .storyboard 结尾的文件，Main.storyboard 和 LaunchScreen.storyboard。Storyboard 是什么呢?

应用一切的界面设计工作都是由 Storyboard 来完成的，它和 Sketch 里的 Artboard 功能是一模一样的，甚至可以理解为我们用 Storyboard 重新复制了一遍在 Sketch 里 Artboard 的设计，目的就是将设计“翻译”成 Xcode“明白”的设计。Storyboard 本身的使用方式和理念也和 Artboard 有很多类似的地方，在之后的实现过程中大家能够体会。

除了界面设计部分，Storyboard 还承担着界面跳转关系以及流程的职责，与使用 inVision 来制作原型和管理设计的方法相类似，也侧面体现了 Storyboard 中“story”的作用。

用 Storyboard 完成设计之后，就能够在手机上体验原生的 Oslo 了，但是这整个过程不需要写任何代码。

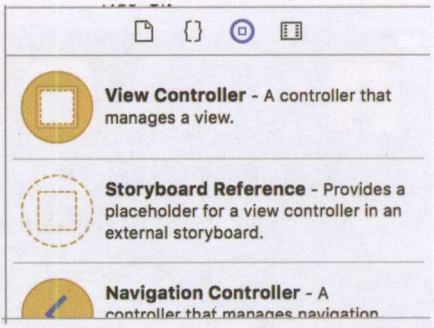
那我们赶快来接触一下 Storyboard 吧。

- 组件 Component -

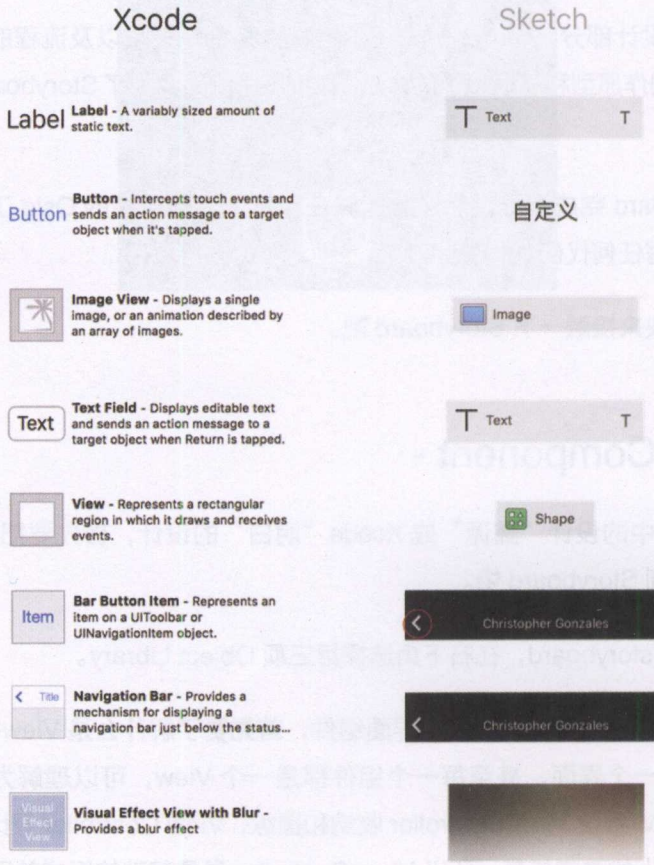
将 Sketch 中的设计“翻译”成 Xcode“明白”的设计，首先要把 Sketch 中的各种元素对应到 Storyboard 中。

单击 Main.storyboard，在右下角选择第三项 Object Library。

这里展示的都是 Xcode 使用的界面组件，首先要了解什么是 View Controller。在 Xcode 中每一个界面，甚至每一个组件都是一个 View，可以理解为都是一个图形。而这些 View 都被 View Controller 收纳和操纵，View Controller 好比一个画布，View 好比画布上的内容信息。所以 View Controller 是最基础的组成单元。



接下来我们对应一下 Sketch 中的设计元素和 Xcode 中的界面组件。



这里需要注意的是：

- Xcode 中的 Label 和 Text Field 都对应 Sketch 中的 Text，这是因为 Sketch 是一个设计软件，目的是用来示意，而在 Xcode 中需要区别其中的功能，不同的功能需要归属于不同的组件。

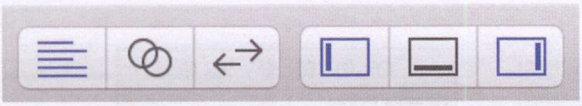
Sketch 中用来画各种形状的 Shape 组件在 Xcode 中对应为 View。但 View Xcode 中组件的基本组成单位，比如 Label、Button、Image View 这些都是 View。

明白了这些对应关系，我们就开始还原应用的第一个界面——Photos。

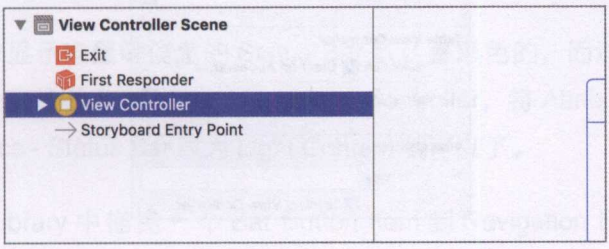
- Table View Controller -

观察 Photos 这个界面，竖向滚动，除了内容不一样以外，其他每一条的样式都是相同的。如果用一个个 View 来实现这些类似的信息，不但写起来累，想必对于机器来说也是比较麻烦的一件事吧……Xcode 为了解决可复用组件的展示问题，根据需要提供了多种 View Controller，它把每一个可复用的组件称为 Cell，我们只需要定义好一个 Cell 的样式，其他 View Controller 对它进行复用就好。这里用到的就是 Table View Controller。

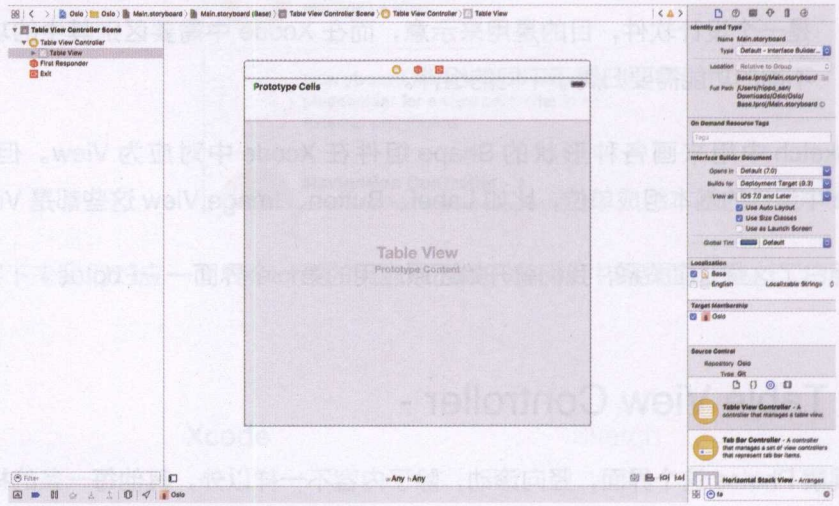
单击右上角从右边数第三个 Hide or show the Navigator(Command+O)，隐藏左侧的 Project Navigator，以节省一些编辑的空间。



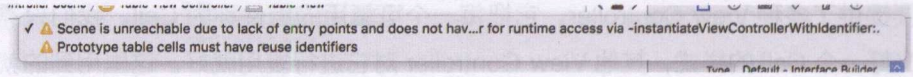
在左侧 Document Outline 中选中 View Controller，按删除键删除。这里不需要 View Controller，需要的是 Table View Controller。



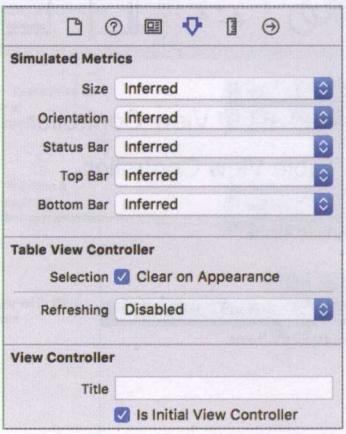
在 Object Library 最下方的 Filter 中输入 ta，会出现 Table View Controller，将它拖曳到中间的 Interface Builder 中。



之后 Xcode 会立刻出现两个错误提示。

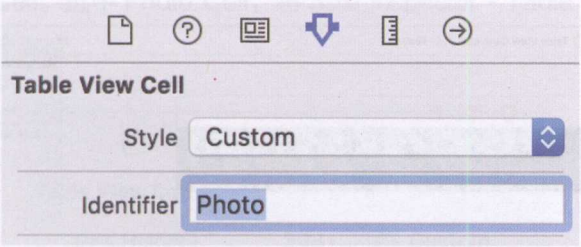


第一个错误提示信息是 Xcode 不知道怎么进入到这个界面。我们需要把它设置为开屏的第一个界面。在 Document Outline 中选中 Table View Controller，在右侧的 Utilities 中选中第四个 Attributes Inspector - View Controller - is Initial View Controller。



这样第一个错误提示就修复了，修复了一个 bug！

第二个错误提示信息是 Xcode 需要对复用的 Cell 有一个识别名。在 Document Outline 中选中 Table View Cell，在 Attributes Inspector - Table View Cell - Identifier 中填写 Photo（Reuse Identifier 命名习惯首字母大写）。



这样第二个错误提示也修复了！

- Navigation Bar -

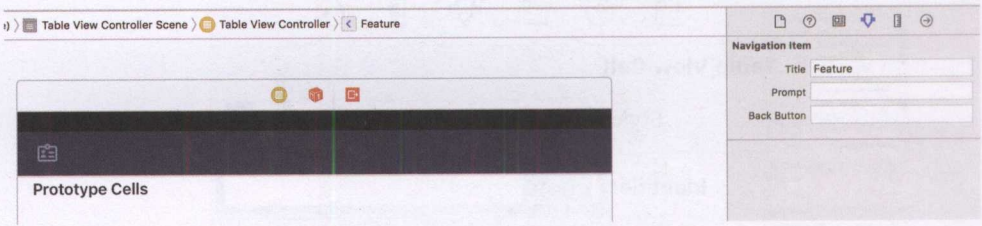
现在开始用 Xcode 从上至下地还原设计，首先实现导航栏。只需要选中 Table View Controller，在菜单栏选择 Editor - Embed In - Navigation Controller，这样 Table View Controller 就被嵌入到 Navigation Controller 中了，也出现了对应的 Navigation Bar。选中 Navigation Bar，将 Attributes Inspector - Navigation Bar - Style 选为 Black，勾掉下方的 Translucent。将 View - Tint 设置为 #FFFFFF，透明度为 80%。

由于最上方显示电量等信息的 Status Bar 默认是黑色的，而这里导航栏也是黑色的，所以会很难看出来。选中 Navigation Controller，将 Attributes Inspector - Simulated Metrics - Status Bar 改为 Light Content 就可以了。

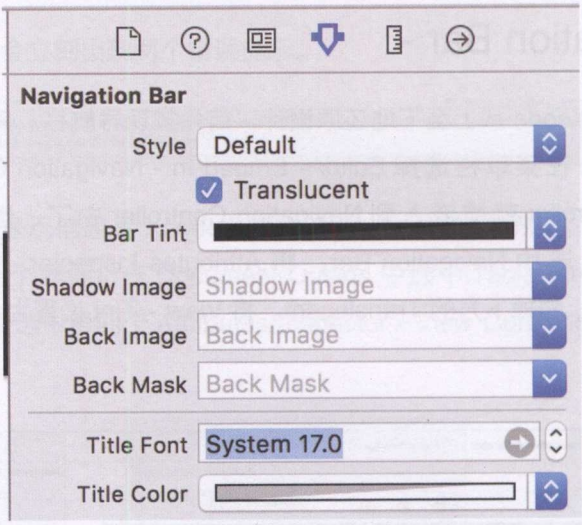
在 Object Library 中拖曳一个 Bar Button Item 到 Navigation Bar 的左侧，在

Attributes Inspector - Bar Item - Image 中填入 user，将 Bar Button Item - Tint 改为设计对应的颜色，删除 Attributes Inspector - Title 中的内容。

选中 Document Outline - Table View Controller - Navigation Item，将 Attributes Inspector - Navigation Item - Title 填入 Feature。

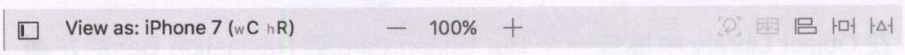


选中 Document Outline - Navigation Bar，将 Attributes Inspector - Title Color 设置为设计颜色，Title Font 设置为设计字体。



导航栏做好后。深有体会的是 Xcode 中 Storyboard 的操作和 Sketch 十分类似，左边是结构总览，中间编辑 Storyboard/Artboard，右边是调节元素属性栏。

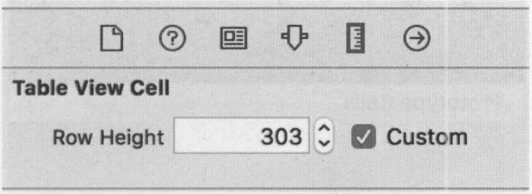
另外会发现 View Controller 的尺寸是默认 iPhone 7 的尺寸，可以在 Interface Builder 下方看到。



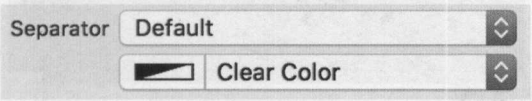
- Table View Cell -

组成 Table View 的单元是 Cell，Cell 可以通过各种组件来自定义样式，甚至是自身的动画，同时也可以通过发送网络请求来获取新的数据，这将在之后的章节介绍。

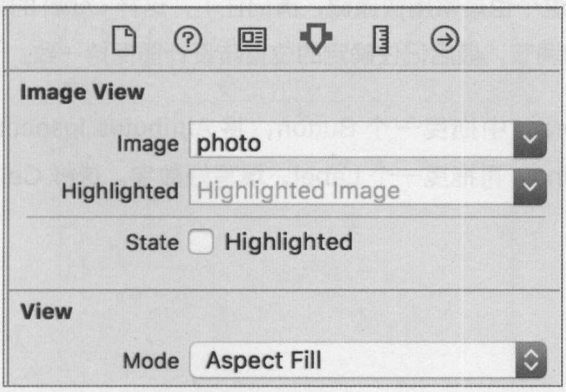
下面来设置 Cell。选中 Photo Cell，将 Size inspector - Table View Cell - Row Height 设置为设计高度。



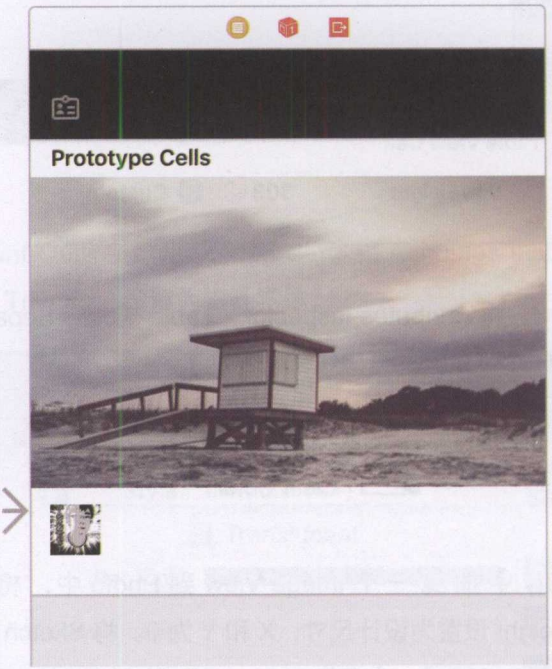
选中 Table View，将 Attributes inspector - Table View - Separator 的颜色设置为 Clear Color，这样 Cell 之间的分隔线就看不到了。



在 Object Library 中拖曳一个 Image View 到 Photo 中，将 Size inspector - View 的 Width 和 Height 设置为设计尺寸，X 和 Y 为 0。将 Sketch 设计里 Unsplash 随机用到的图片导出 .pdf 格式，命名为 photo.pdf，放入 Assets.xcassets 内。回到 Image View，将 Attributes Inspector - Image View - Image 设置为 photo，View - Mode 选择 Aspect Fill 来保证画面的比例不失衡并且损失的部分也相对小。



按照设计图，以同样的方法建立用户头像（先不用在意是方形还是圆形），勾选 Attributes Inspector - Drawing - Clip Subviews，保证图片不超出规定范围大小。间距需要跟设计图保持一致，可以在 Sketch 文件中按住 Alt 键在元素间移动鼠标来确认间距，在 Storyboard 中也可以利用 Sketch 里的方法来查看和移动组件位置。

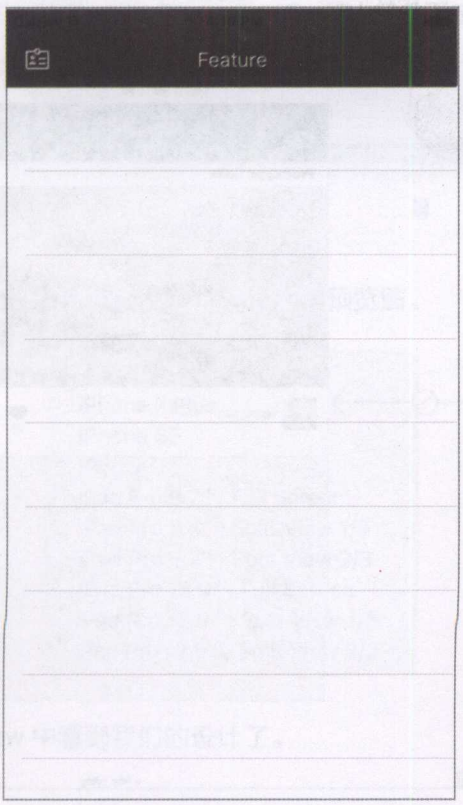


在 Object Library 中拖曳一个 Label，在 Attributes inspector 中输入相应的内容，选择与设计图对应的字体后，在菜单栏单击 Editor - Size to Fit Content（Command+=，这个也是常用快捷键，请记住），这样 Label 的大小就完全贴合于内容了，不用手动调节，最后记住确定的位置与设计图保持一致。

在 Object Library 中拖曳一个 Button，将 Attributes inspector - Background 设置为 heart-outline。再拖曳一个 Label，填写好数字。这样 Cell 的设计还原就完成了。

- Preview -

既然还原完设计了，赶快 Build 一下吧。



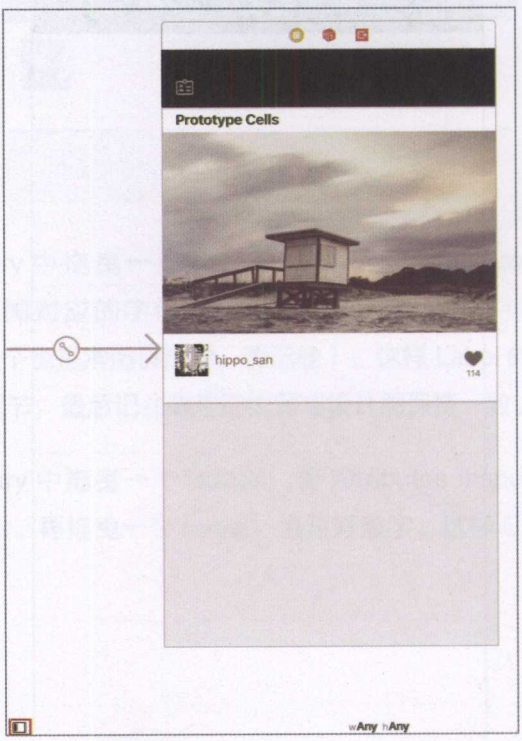
在 Simulator 中看到的只有 table view 结构的空白格子，但实际上在 Storyboard 中，table view 可以显示静态数据和动态数据两种，静态数据即为我们这种“写死”了的图片和 Label 之类的元素内容，动态数据是服务端返回数据更新的元素内容。table view 默认使用的是动态数据，然而我们前面没有做展示动态数据的这部分内容，所以没有显示。

为了确认设计成果，还有一个方法，即使用 Xcode 的 Preview 功能与 simulator 相比，Preview 有以下几个好处。

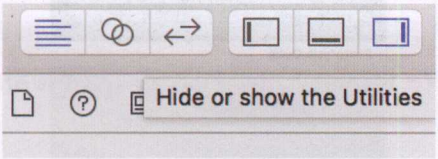
- 可以显示静态数据的设计。
- 在不用 Build 的前提下实时反映编辑后的变化。
- 支持多种 iOS 设备的显示结果。

基于这些，我们应该在设计过程中更多地使用 Preview。

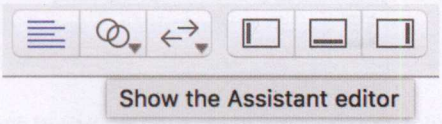
首先单击 Editor 左下角的 Hide Document Outline 节约一些空间。



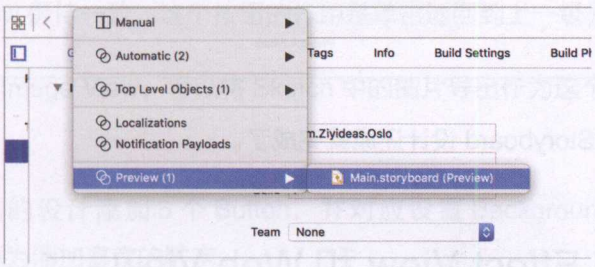
然后单击屏幕右上角的 Hide or show the Utilities。同样为了节省更多可视空间单击下图最右方按钮。



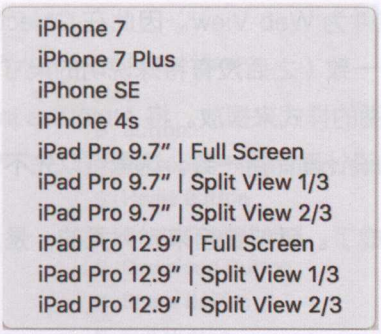
单击第二个 Show the Assistant editor。



在新出现的界面左上角位置找到 Manual，选择最后一个选项 Preview - Main.storyboard（Preview）。



通过界面左下角的“+”，来添加 iPhone 7 的预览图。



然后就能在 Preview 中看到我们的设计了。

这样首页的 Storyboard 设计还原就完成了。

- Visual Effect View 和 Web View -

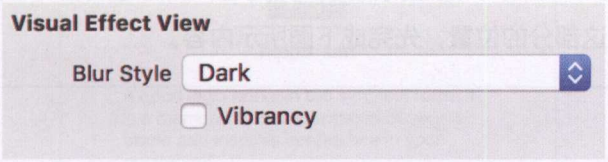
介绍还原 Login 界面的设计。Login 界面由一个内嵌的网页和一个返回按钮组成。在 iOS 中，内嵌网页的组件为 Web View。因此在 Object Library 中拖曳一个 Web View，尺寸和设计图保持一致（之后没有特殊说明的尺寸均为设计图尺寸）。再拖曳一个 Button，按照设计图的样式来摆放。将 Attributes inspector - Button - Title 设置为 Back，字体和颜色和设计图保持一致。这里可以先不管 Button 圆角的问题。

这样 Login 界面就完成了。目前看起来光秃秃的，是因为还没有内容嘛，下面我们一点点增加。

其实还有一个界面我们也可以用 Web View 来实现，那就是 Profile 界面中单击用户的 Instagram 或者个人网站后的页面。这里可以再拖曳一个 View Controller，然后在上面加一个 Web View，暂时存放在这里。其实我想说的是，你看，有的时候

不需要出设计图，仅靠组件本身就可以完成一个功能！

下面还原 Photo 界面的设计。在 Object Library 中拖曳一个 View Controller。再拖曳一个 Visual Effect View with Blur 到 View Controller 中，尺寸和设计图保持一致，暂时忽略 Xcode 的黄色错误提示。在 Attributes inspector - Visual Effect View - Blur Style 中选择 Dark。



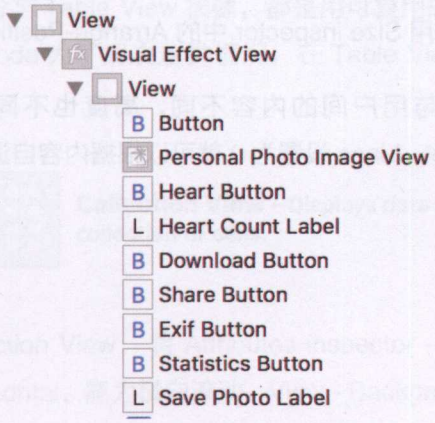
这样一个带有模糊效果的视图就建成了。在运行应用的时候，只要在这个视图之下的视图，都会做模糊处理，十分方便。

在 Object Library 中拖曳一个 Button 到 Visual Effect View with Blur 上，尺寸和 Visual Effect View 保持一致，这个按钮的作用是单击返回到上一级界面。

再拖曳一个 Image View，可以将 Sketch 中的图片导出作为这个 Image View 的 Image。

按照设计图的设计添加 5 个 Button，并对应设置 Background。再拖曳一个 Label，内容信息为添加喜欢的数字。

这样 Photo 界面的还原也完成了。但需要注意的是组件的层级结构，如下图所示。



这里可以先不用像上图那样为组件命名，之后的部分组件可以自动进行命名。

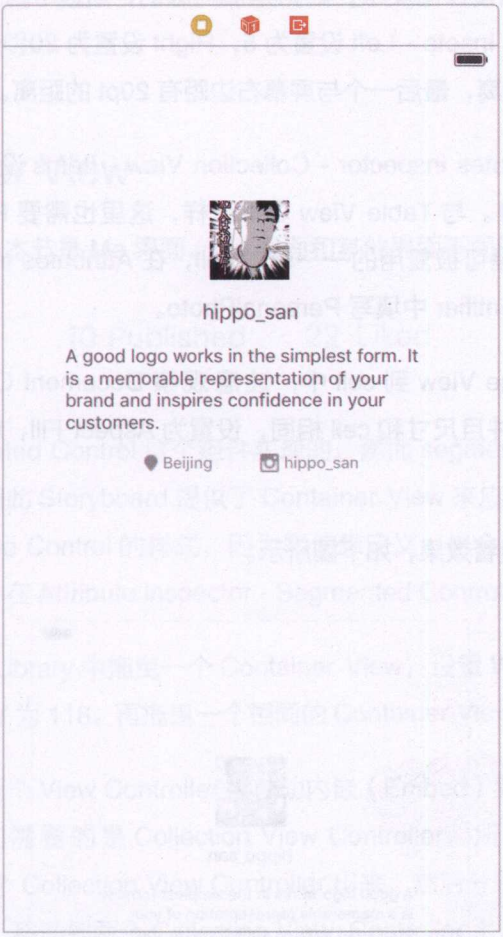
- Collection View -

下面介绍 Profile 界面。还是先观察后设计，发现这个界面也具有导航栏（Navigation Controller），这里我们先不做处理，因为之后会有很方便的方法能够快速完成。留出这部分的位置，先完成下图所示内容。

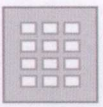
制作过程就不重复了，相信现在的你已经足够熟悉组件了，下面是几个小提示。

- 使用的组件包括 View Controller、Image View 和 Label。
- 对齐方式可以使用 Size inspector 中的 Arrange-Position in Container。
- 用户简介部分与用户间的内容不同，高度也不同，只需要将 Attribute inspector - Label - Lines 设置为 0 就可以根据内容自适应高度了。

完成后如下图所示。



接下来图片的部分与 Table View 很像，都是用可复用的 cell 来展示的，只不过这里是横向滚动。Xcode 为了解决这种情况，在 Table View 之外，还提供了一个 Collection View 组件。



Collection View - Displays data in a collection of cells.

拖曳一个 Collection View，将 Attributes inspector - Collection View - Scroll Direction 设置为 *Horizontal*，意为横向滚动。*View* - *Background* 设置为 *#FFFFFF*。选择 Size inspector - View 按照设计图设置尺寸，这里的 Width 为屏幕最大宽度，即

375。Cell Size 按照设计图尺寸，因为是横向滚动，所以 Min Spacing - For Lines 设置为 20，Section Insets - Left 设置为 5，Right 设置为 20，保证第一个 cell 和屏幕左边距有 5pt 的距离，最后一个与屏幕右边距有 20pt 的距离。

还可以将 Attributes inspector - Collection View - Items 设置为 2，确定设计样式没问题后再改回 1。与 Table View Cell 一样，这里也需要 Reuse Identifier 来告诉 Xcode 这个 cell 是可被复用的——单击 cell，在 Attributes inspector - Collection Reusable View - Identifier 中填写 PersonalPhoto。

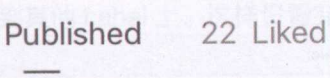
拖曳一个 Image View 到 cell 中，注意观察 Document Outline，确认 Image View 是在 cell 中，并且尺寸和 cell 相同，设置为 Aspect Fill，Image 可以继续沿用之前的 photo。

在 Preview 中查看效果，如下图所示。

这样 Profile 页面基本完成了，至于里面导航栏的设置，头像的圆角和图片的阴影，在后面的部分分别说明。

- Container View -

继续还原工作，本节是 Me 界面。Me 界面和其他界面不同的部分如下图所示。



是通过 Segmented Control 这个组件实现的，然而 segment 的切换，也导致了下方视图的切换。因此 Storyboard 提供了 Container View 来应付这种情况。先不要着急定义 Segmented Control 的样式，因为即使想定义，也会发现定义不了。应先改这里的文字内容，在 Attribute inspector - Segmented Control - Title 中可以更改。

接着从 Object Library 中拖曳一个 Container View，设置 Width 为 211，Height 为 549，X 为 164，Y 为 118。再拖曳一个相同的 Container View 覆盖在之前的上面。

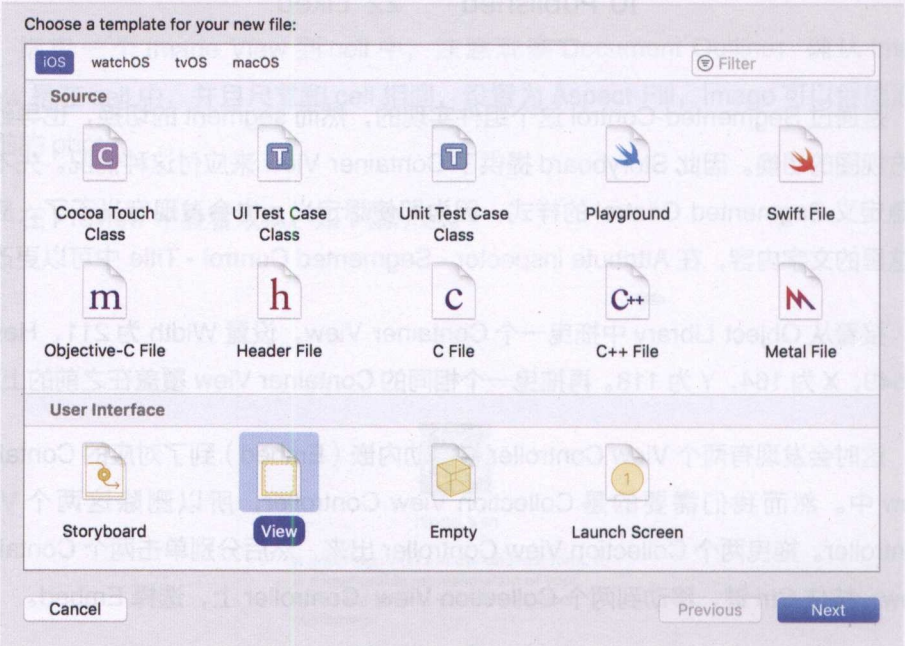
这时会发现有两个 View Controller 被自动内嵌（Embed）到了对应的 Container View 中。然而我们需要的是 Collection View Controller，所以删除这两个 View Controller。拖曳两个 Collection View Controller 出来，然后分别单击两个 Container View，按住 Ctrl 键，移动到两个 Collection View Controller 上，选择 Embed。

在两个 Collection View Controller 的 cell 上再分别添加一个 Image View。

- Stack View -

距离我们的还原工作结束还差两个界面——单击 statistics 和 camera 后出现的界面。在 Xcode 中除了 Storyboard 可以还原界面，还可以使用 View 来还原。我们先还原单击 camera 后的界面。

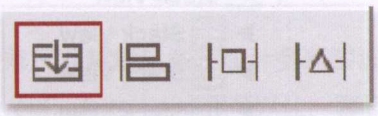
单击 File - New - File，选择 iOS - User Interface - View - Next。



文件命名为 ExifView.xib。选中 DocumentOutline 中的 View，将 Attribute inspector - SimulatedMetrics - Size 设置为 Freeform。再在 Size inspector 中设置 Width 为 345，Height 为 363。

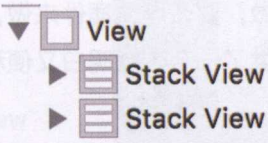
接下来设置各种文字信息。可以发现文字信息很多，布局也有相似之处，针对这种情况，Xcode 提供了一个方便的组件——Stack View，用于方便地解决多个组件对齐和布局的问题。Stack View 也能够大大缩短 Auto Layout 的时间，这一点将在后面讲到。

首先将每一个标题及所有的信息放到一个 Stack View 中。比如按住 Shift 键同时选中 Published 和日期，然后单击右下方最左侧的 stack。

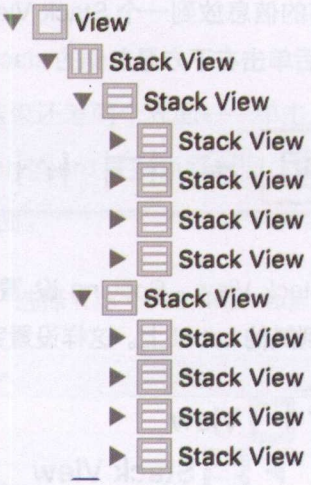


将 Attribute inspector - Stack View - Spacing 设置为 6，Alignment 设置为 center。使用同样的方法应用到其他 Label 上。这样设置完应该是 8 个 Stack View。

再将左列的 4 个 Stack View 外面再设置一层 Stack View，将 spacing 设置为 20。右列重复这个步骤。



最后将这两个 Stack View 外面再设置一层 Stack View，将 spacing 设置为 6。



这样就完成了界面的设计还原。

对于有规律的组件布局，Stack View 帮了大忙，而且 Stack View 能够根据元素现有的位置自动调节是横向的还是纵向的。

设计下一个 StatisticsView 也是同样的道理，赶快用 Stack View 试试吧。

完成后，Sketch 中的设计就全部还原了，可喜可贺！

设计都还原出来后，但是这些只保证在 iPhone 7/7 Plus 下的显示效果，如果这时在 Preview 中加一个 iPad，就会出现错误显示。我们需要做一款 Universal App，在所有 iOS 设备上都可以正常显示。这在 Sketch 的设计阶段完成是比较麻烦的，要么使用新尺寸的 Artboard 来做，要么借助插件来做，但无论使用哪种方法，似乎都笨重一些。在 Xcode 中，提供了一个容易明白又便利的工具来解决这个问题——Auto Layout。

Auto Layout

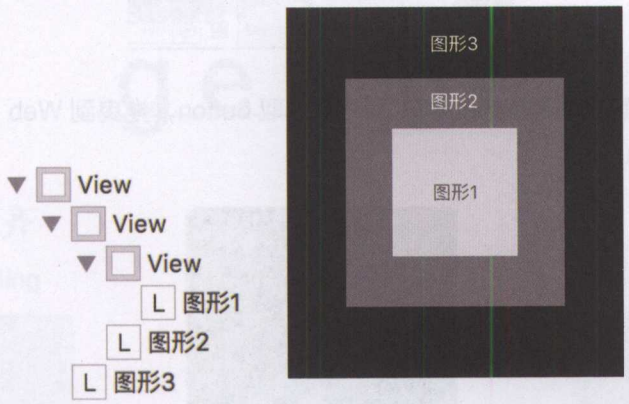
随着 Apple 推出的设备尺寸越来越多，设计多兼容设备已经变得十分重要，因此 Xcode 引入了 Auto Layout。Auto Layout 可以让一套设计在任何 iOS 设备上都保持显示一致，并且还可以自定义在特定设备上的显示。

若懂得 Web 开发中的 CSS（层叠样式表）的话，则可以把 Auto Layout 看作是 CSS 中 margin、padding、position 这些定位属性的集合。比如定义了两个元素间的距离是 10pt，那么在所有设备上的显示都是 10pt；定义了元素垂直居中显示，那么在所有设备上都是垂直居中显示。

那我们就从个简单的界面入手，开始探究 Auto Layout 吧。

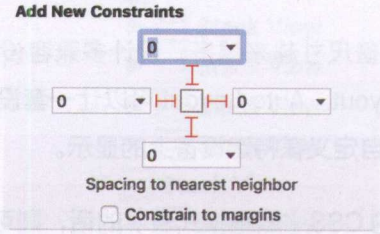
- 对齐 & 间距 Alignment& Spacing -

首先从相对容易的 Login 界面入手。先复习一下 Document Outline 中的层级结构和 Storyboard 的对应关系，如下图所示。



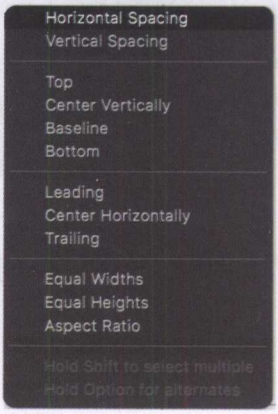
选中 Web View，单击下方右侧 Add New Constraints 按钮。

选中上下左右，距离均设置为 0，这样就保持了在任何设备上 Web View 的尺寸都和设备屏幕尺寸一致，但需要取消勾选 Constrain to margins 项。



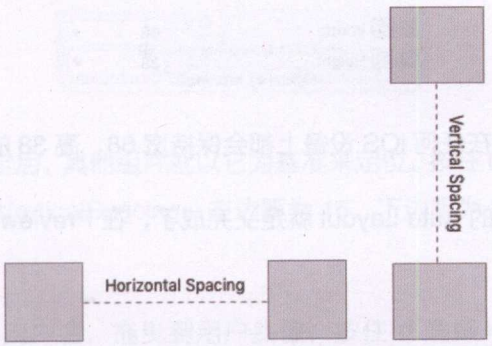
接着选中 Button，单击下方的 Align 按钮，选择 Horizontally in Container 保持水平居中。

按住 Ctrl 键，在 Document Outline 中选取 button，拖曳到 Web View，会出现如下图所示菜单。

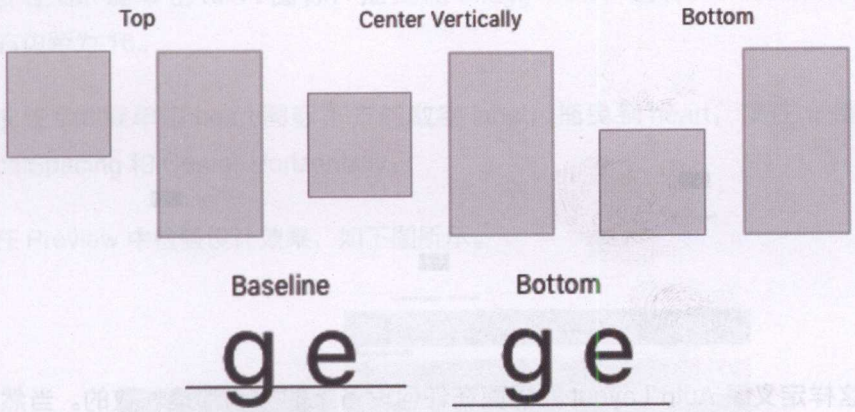


这种操作方式定义了两个组件的位置关系，如下图所示。

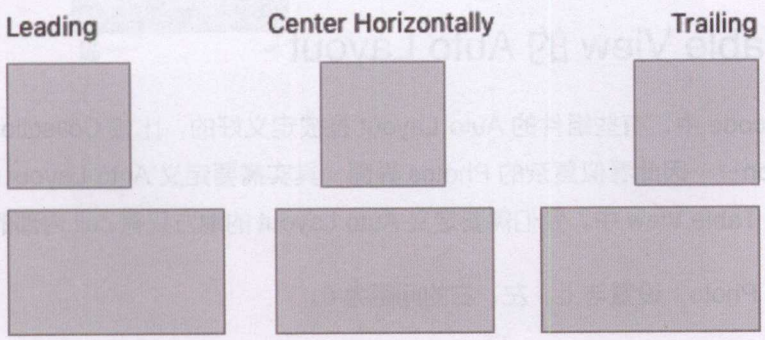
距离



竖向对齐



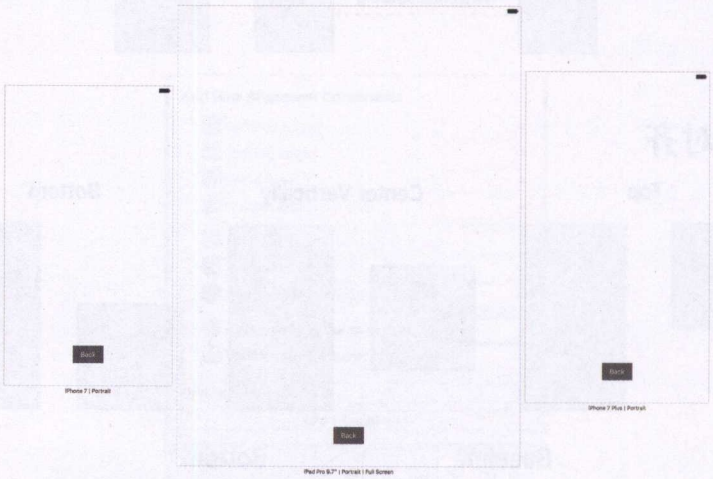
横向对齐



单击 Vertical Spacing，定义 button 和 Web View 的竖向距离。这时 Xcode 会出现报错，我们还需要定义组件的大小。再次单击 Add New Constraints，勾选 Width 和 Height，填写设计值。

这样组件的大小在任何 iOS 设备上都会保持宽 68，高 38 的大小。

这样 Login 界面的 Auto Layout 就定义完成了，在 Preview 中检查一下吧。



这样定义了 Auto Layout 的界面在任何设备上的显示都是一致的。当然 Auto Layout 远不止这些内容，下面我们来慢慢探索。

- Table View 的 Auto Layout -

在 Xcode 中，有些组件的 Auto Layout 是被定义好的，比如 Collection View、Bar Button…… 因此看似复杂的 Photos 界面，其实需要定义 Auto Layout 的地方并不多。在 Table View 中，我们需要定义 Auto Layout 的地方只有 cell 内部的内容。

单击 Photo，设置与上、左、右的间距为 0。

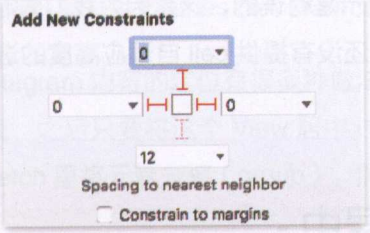


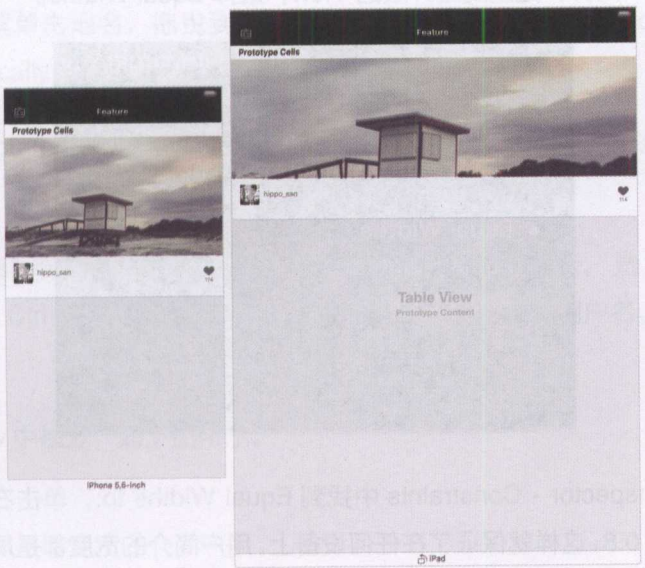
Photo 的定位确定后，其他组件就以它为基准来定位。按住 Ctrl 键单击用户头像，拖曳到 Photo，选择 VerticalSpacing。左边距为 16，下边距为 29，并勾选 Width 和 Height。

按住 Ctrl 键单击用户名，拖曳到用户头像，按住 ⇧ 选择 HorizontalSpacing 和 Center Vertically 后按 Enter 键。

按住 Ctrl 键单击 heart 图标，拖曳到 Image View，选择 VerticalSpacing 为 16。右边距为 16。

按住 Ctrl 键单击 heart 图标下方的数字 label，拖曳到 heart，按住 ⇧ 键选择 VerticalSpacing 和 Center Horizontally。

在 Preview 中检验设计效果，如下图所示。



上图 iPad 中图片的显示是有误的，这是因为我们限制了 cell 的高度，因此出现了问题。在 Storyboard 里还没有提供 cell 自适应高度的选项，这个问题后面会用其他方式来解决。

- 多个元素的居中 -

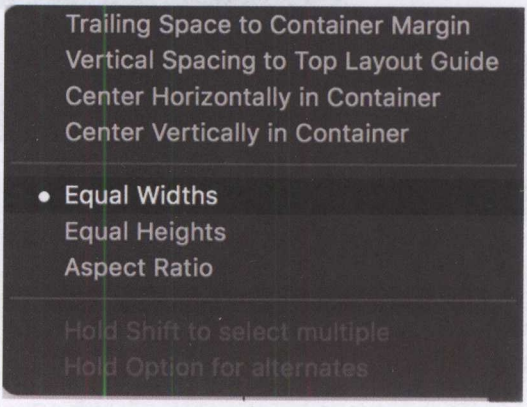
下面来做 Profile 界面的 Auto Layout。

按住 ⇧ 键选中用户头像、用户名和用户简介，在 Auto Layout 的 Align 选择 Horizontally in Container，使这三个组件水平居中于屏幕显示。

单击用户头像，选择 Vertically in container。在 Size inspector - Constraints 中找到 Align Center Y to:，单击右侧的 Edit。将 Multiplier 改为 0.42，这样就保证了在任何设备上，用户头像的垂直位置都是屏幕中心 × 0.42 的位置。然后勾选 Width 和 Height。

用户名和用户简介的垂直位置都以用户头像为参考。按住 Ctrl 键单击用户名，拖曳到用户头像，选择 VerticalSpacing。

按住 Ctrl 键单击用户简介，拖曳到用户名，选择 VerticalSpacing。再次按住 Ctrl 键单击用户简介，拖曳到最外层的 View，选择 Equal Widths。

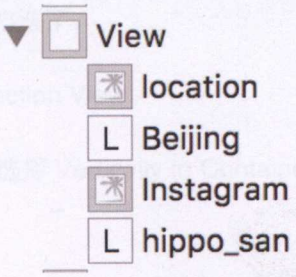


在 Size inspector - Constraints 中找到 Equal Widths to:，单击右侧的 Edit。将 Multiplier 改为 0.8，这样就保证了在任何设备上，用户简介的宽度都是屏幕宽度 × 0.8。

如果有黄色的错误提示，可以 Update Frames (Alt + Command + =)。

接下来 location 和 Instagram 内容的定位有很多种做法。有一种办法是将下面这个部分包含在一个 View 里，之后只要将这个 View 居中对齐，其他组件相对于它来定位就可以了。类似于 Sketch 里将元素分组（group），然后来调整 group 的定位。

拖曳一个 View，在 Document Outline 中确认层级关系如下图所示。



根据 Sketch 中的 Sub info 尺寸来设置这个 View。按住 Ctrl 键单击它，拖曳到用户名，选择 VerticalSpacing。在 Auto Layout 的 Align 选择 Horizontally in Container。

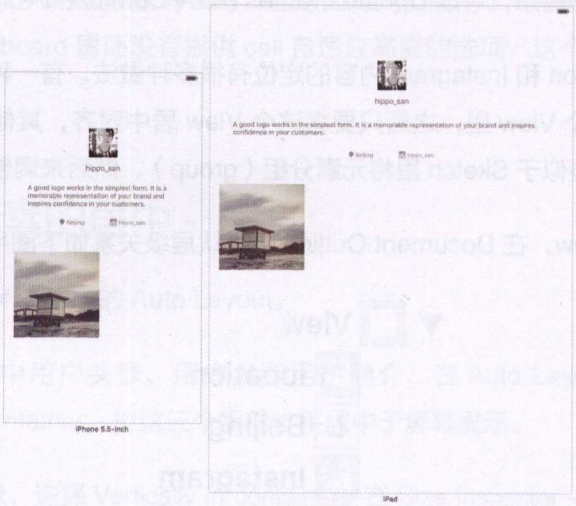
单击 location 图标，在 Auto Layout 的 Align 选择 Vertically in Container。设置左边距和上边距为 0。

按住 Ctrl 键单击地名，拖曳到 location 图标，按住 ⇧ 选择 Horizontal Spacing 和 Center Vertically。

单击 Instagram 用户名，设右边距为 0。在 Auto Layout 的 Align 选择 Vertically in Container。

按住 Ctrl 键单击 Instagram 图标，拖曳到 location 图标，选择 Horizontal Spacing。按住 Ctrl 键单击 Instagram 图标，拖曳到 Instagram 用户名，选择 Center Vertically。

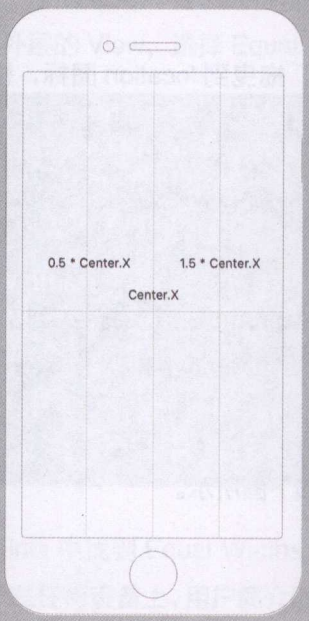
在 Preview 中检查，如下图所示。



检查完毕，没有问题！

第二种方法

第二种方法是利用 Contraints 里的 Multiplier。比如实现“这个元素应该位于屏幕中间左侧的 50%”，可以先找到屏幕中部，然后乘以一个乘数。



计算一下这个 Multiplier：107（location 图标距离左侧屏幕边缘距离）/（375/2）（屏幕的一半宽度）=0.5707。乘数计算出来以后，Auto Layout 设置 Center Horizontally in Container，单击标注线，在 Size inspector 中将 Multiplier 设置成这个乘数就可以了。

还可以用 Stack View 来完成，能更快地实现多元素的居中对齐，不过这里是为了介绍概念，就不使用这种方法了。

下面来设置最后的 Collection View。

选中 Collection View，选择 Vertically in Container，将 Multiplier 设置为 1.48。设置左右边距都为 0。

按住 Ctrl 键单击 Collection View，拖曳到最外层的 View，选择 Equal Heights，将 Multiplier 设置为 0.3。

选择 Collection View 中的 Image View，设置上下左右边距均为 0，使图片充满整个 cell。

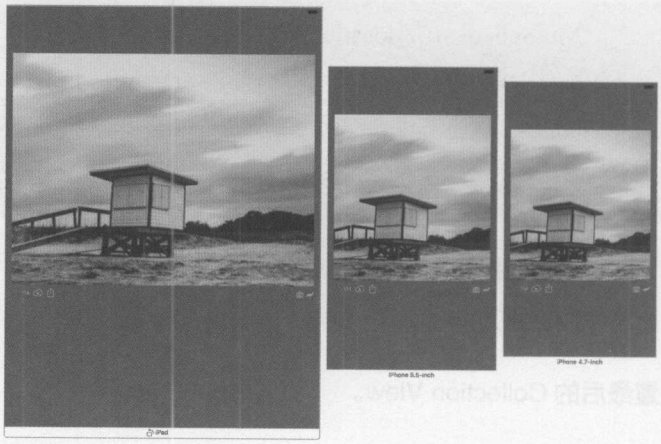
这样 Collection View 的 Auto Layout 就完成了，整个 Profile 页面也完成了。

- 用 Photo 界面练习 -

使用 Photo 页面作为练习，需保持原图的长宽比，这样在不同尺寸的设备上都可以保证 photo 的原图比例，只是图片大小不同（注意活用 Equal Widths/Heights），但能够最大限度地展示图片。

接下来介绍的 ExifView 和 StatisticsView，只需设置最外层 Stack View 的 Auto Layout，让它水平或垂直居中就可以了，感谢 Stack View。

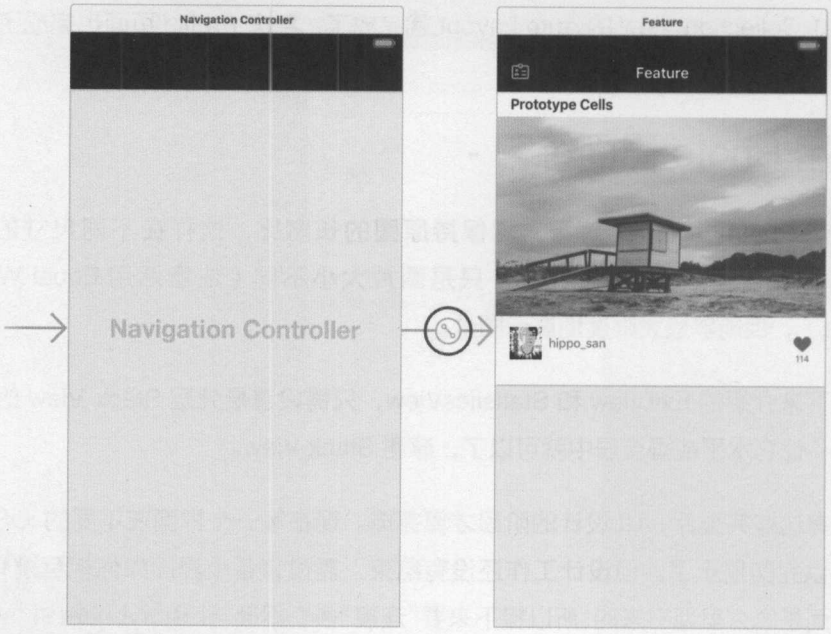
所有这些完成后，UI 设计的阶段才算完成。现在每一个界面在不同的 iOS 设备上都可以完美显示了。但设计工作还没有结束，需设置各个界面如何相互跳转，及每个界面是怎么呈现出来的。所以接下来要“连接”各个界面，让 Storyboard 讲“story”。



Segue [sey-gwey, seg-wey]

Segue 源为意大利语，意思是跟随。在 iOS 中，Segue 是连接界面和界面之间 (View Controller 之间) 的纽带，它表示两个界面是如何相连以及过渡的。

其实我们已经看到了一个 Segue 了，在将 Photos 界面内嵌在 Navigation Controller 中时，一个 Segue 就产生了，如下图所示。



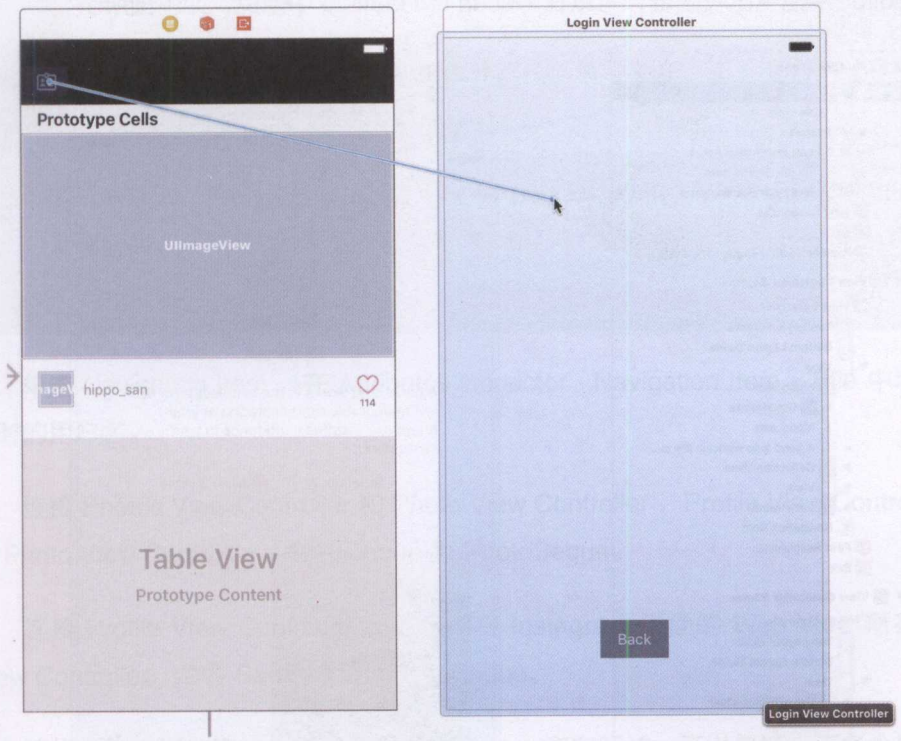
Segue 有不同种类，可以参考 https://developer.apple.com/library/ios/recipes/xcode_help-IB_storyboard/Chapters/StoryboardSegue.html Table 1 部分。

Segue 不但可以连接各个 Storyboard ，同时还可以传输不同界面之间的数据，控制界面出现时机和动画等。

- 连接 Storyboard -

下面用 Segue 来连接其他界面。

按住 Ctrl 单击 Photos 界面中的 user 图标，拖曳到 Login View Controller 上，如下图所示。

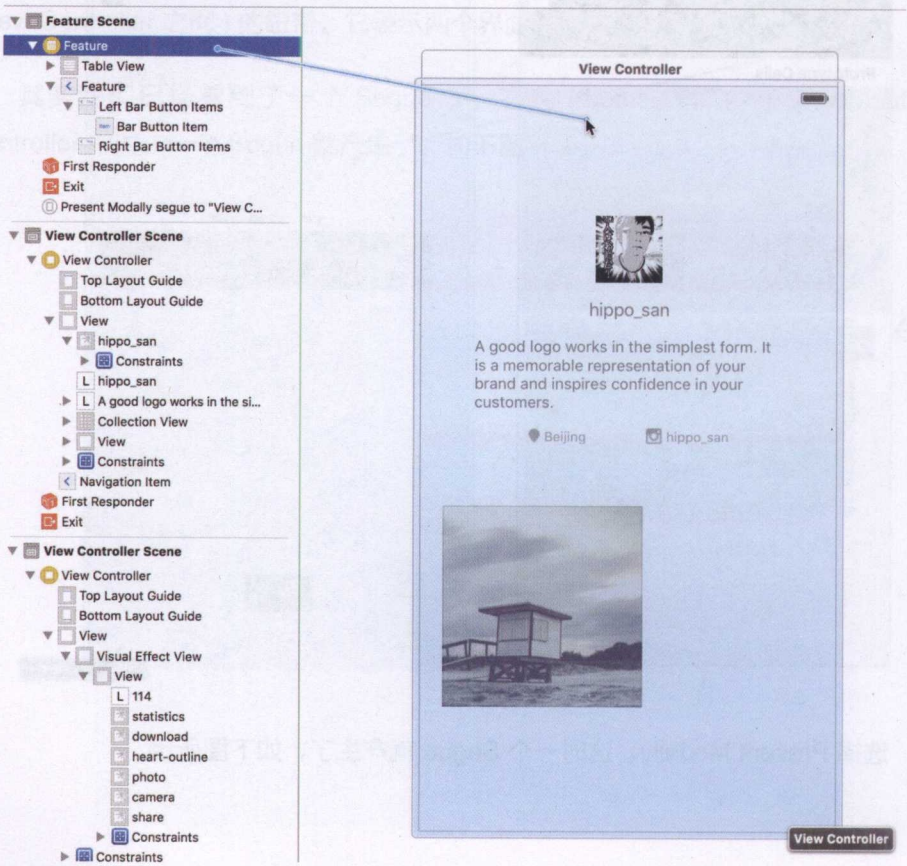


选择 Present Modally。这时一个 Segue 就产生了，如下图所示。

单击这个 Segue，在 Attributes inspector 中，将 Identifier 填入 LoginSegue (Segue 命名习惯首字母大写)，在 Presentation 中选择 Over Current Context，在 Transition 中选择 Cross Dissolve，这就定义了被连接的 Login 界面如何展现出来。

这里先暂时不用管 Me 界面的连接。

按住 Ctrl 单击 Photos View Controller，拖曳到 Profile View Controller，如果在 Editor 中拖曳比较困难，可以在 Document Outline 中拖曳，如下图所示。



选择 Show。之后可以看到 Profile 界面中自动出现了导航栏，这就是之前说的快捷建立导航栏的方法，因为 Profile 是“show”出来的，所以在层级关系上为 Photos 的下一级。Segue 命名为 ProfileSegue。

但也因为多出了导航栏，页面上已经存在的组件定位都有了偏差。只要在错误提示中将每个错误都“Fix Misplacement”就可以了。

为什么不用 Photos 页面的用户头像 / 用户名来连接 Profile View Controller？

你会发现即使想连接也是无法连接成功的。不是所有的组件都能连接 View Controller。一般情况下只有 Button 和 View Controller 可以连接。所以之前的 Bar Button 是可以直接连接的。

因此之外的组件连接到 View Controller 有两种做法：

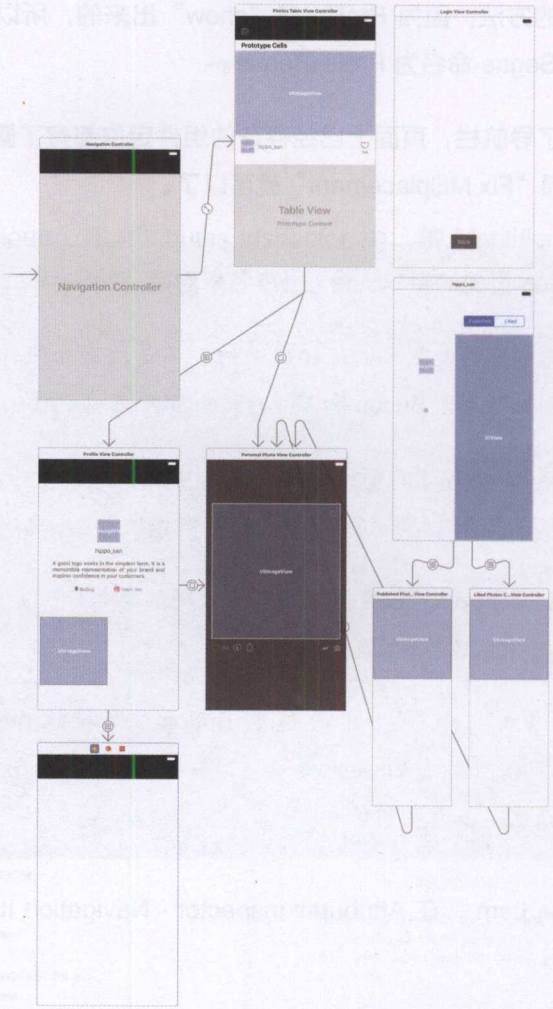
- 将两个 View Controller 相连，在之后的代码编写部分，定义专门的地方来触发连接好的 Segue，实现页面跳转。
- 在组件上建立一个等尺寸的透明 Button，连接这个 Button 和 View Controller。这个方法的缺点是不但要定义这个透明 Button 的 Auto Layout，还产生了多余的组件。

单击 Navigation Item，在 Attributes inspector - Navigation Item - Title 中填入暂时的用户名。

连接 Photos View Controller 和 Photo View Controller，Profile View Controller 和 Photo View Controller，命名 Segue 为 PhotoSegue。

连接 Profile View Controller 和展示用户 Instagram 页面的 Web View 所在的 View Controller，命名 Segue 为 PortfolioSegue。

这样界面之间的连接就完成了。双击 Editor 空白区域，可以总览一下各个界面之间的关系，如下图所示。



至此，设计的部分就结束了。可以看到设计不只包含 Sketch 的 UI 设计，还包含原型的制作，甚至包含可能的动画设计。

但是只在 Preview 里看到设计还远远不够，在真机上能够把玩到真实的原型、看到真实的数据才算是一款合格的应用。好了，现在要进入代码的世界了，一步一步来，代码远比想象的简单！

编程

用 Swift 3 来编写规范的代码。

二

Swift 介绍

Apple 旗下的产品代码在过去都是用 Objective-C 来编写的，直到 2014 年发布了新的 Swift 语言后，全线产品代码就自然转到了使用 Swift 上来。并且 Swift 在不断地创新演化，到目前已经发展到 3.0.2 版本。

本书不介绍如何书写 Swift，而是从完成设计作品的角度出发，由浅入深，推荐学习以下几个资料。

- Playgrounds：如果你拥有 iPad，在更新到 iOS 10 后会出现一个名为 Playgrounds 的应用，它是 Apple 官方开发的采用游戏互动的方式学习 Swift 的应用。以这里为起点开始学习，有趣又直观。

- The Swift Programming Language：Apple 官方出的 Swift 指南，有详细的语法介绍和示例代码。可以在 iBooks 上搜索下载。

- Getting Started with Swift：WWDC 2016 介绍了 Swift 基础知识（https://developer.apple.com/videos/play/wwdc2016/404/），贯穿了大部分的概念，浅显易懂。

- Guides：在掌握了 Swift 的基本语法和知识后就可以进行实际操作了。可以从 Apple 官方出的 Guides（在 https://developer.apple.com/library/prerelease/content/navigation/ 中搜索 Guides），跟着一步一步练习学习。

- Sample Code：Guides 帮助大家掌握基本的编写思想和结构，Sample Code 则从实例出发，以实战的方式帮助大家认识代码。

以上资料都学习掌握后，就基本可以开发自己的作品了。如果遇到困难，也可以借助类似 http://stackoverflow.com/ 这样专业的技术网站来寻求帮助。

先把这些准备工作做好，然后开始 Oslo 接下来代码部分的制作。

用代码控制界面

- 关联 Storyboard 和代码文件 -

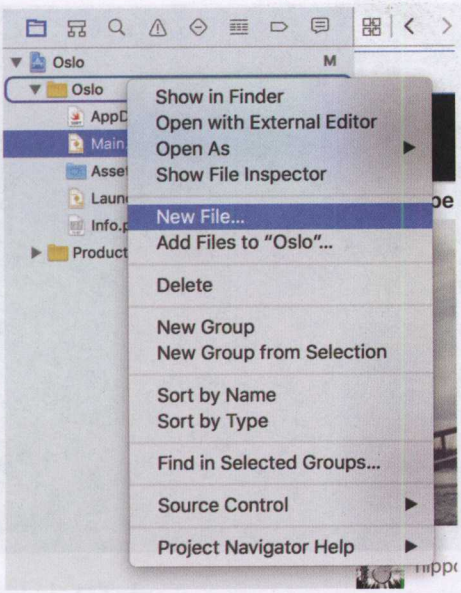
如何做到用代码控制组件呢？在 Xcode 中是按下面这几个步骤做的：

1. 建立一个写代码的“地方”。

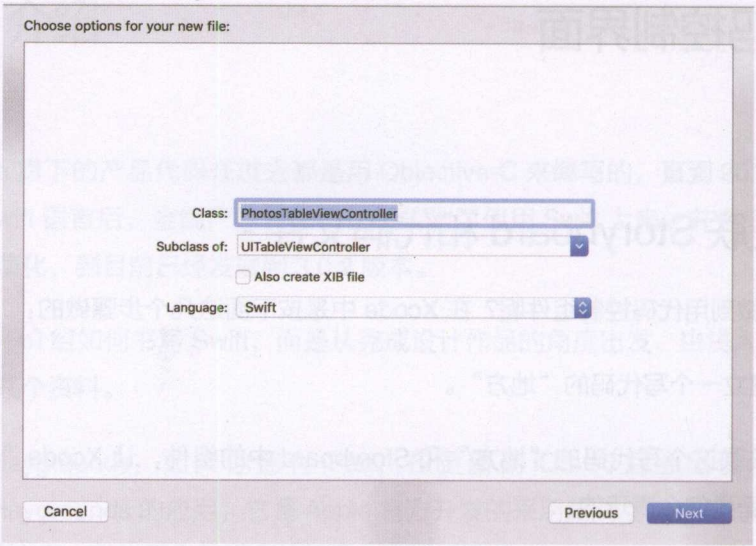
2. 关联这个写代码的“地方”和 Storyboard 中的组件，让 Xcode“明白”组件在代码里是怎么表示的。

在 Project Navigator 中删掉 ViewController.swift，删除的时候选择 Move to Trash。

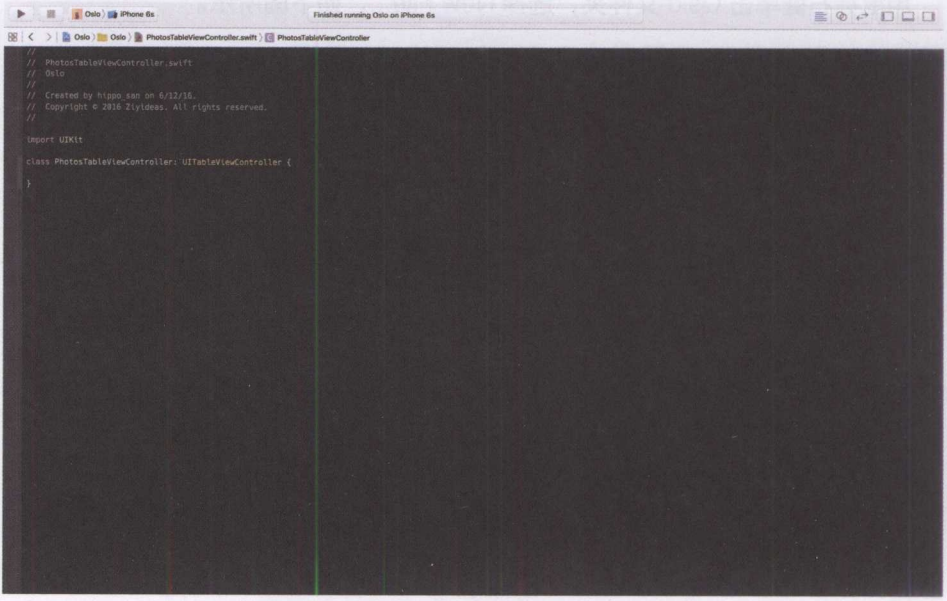
用鼠标右键单击 Oslo 文件夹，选择 New File...，如下图所示。



在哪个地方单击右键，建立的文件就会在哪个地方的下方。选择 iOS - Source - CocoaTouchClass，单击 Next，在 Subclass of 中填写“UIT”后会自动填充为 UITableViewController，在 Class 中补全为 PhotosTableViewController。如下图所示。



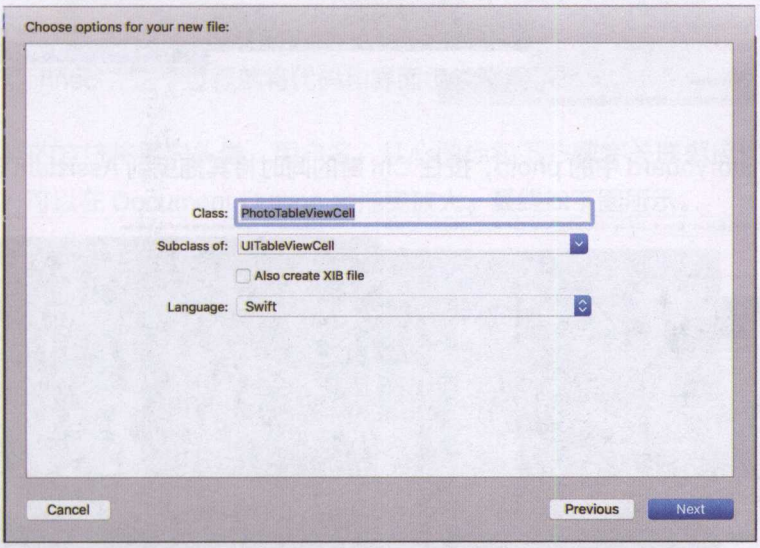
单击 Next。再单击 Create。然后单击创建好的 PhotosTableViewController.swift，删除最外层大括号里的内容后如下图所示。



或许你看到的界面和我看到的不一样，因为 Xcode 代码编写的地方是可以自定义主题的，如果你喜欢我使用的主题，可以在 https://github.com/chriskempson/tomorrow-theme 获得。

写代码的“地方”建立好后，就要和 Storyboard 中的组件关联起来。选择 Photos View Controller，在 Identity inspector - Custom Class - Class 中填入 p，会自动补全为 PhotosTableViewController，按回车键。这样就关联好代码控制和界面两个地方了。

因为 cell 是自定义的，所以 cell 也需要建立一个代码控制的“地方”。用同样的方法，右键单击 PhotosTableViewController.swift，创建一个名为 PhotoTableViewCell 的文件。如下图所示。



同样删除最外层大括号里面的内容。

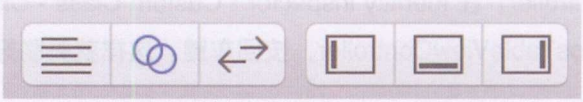
回到 Storyboard，选择 PersonalPhoto，在 Identity inspector - Custom Class - Class 中填入 p 后会自动补全为 PhotoTableViewCell，然后按回车键。

这样 Xcode 就“明白”代码控制和界面的关系了，下面需要做的是把界面里的组件拖曳到代码控制的“地方”，这样才能够通过代码来控制界面组件。

- 连接组件到代码中 -

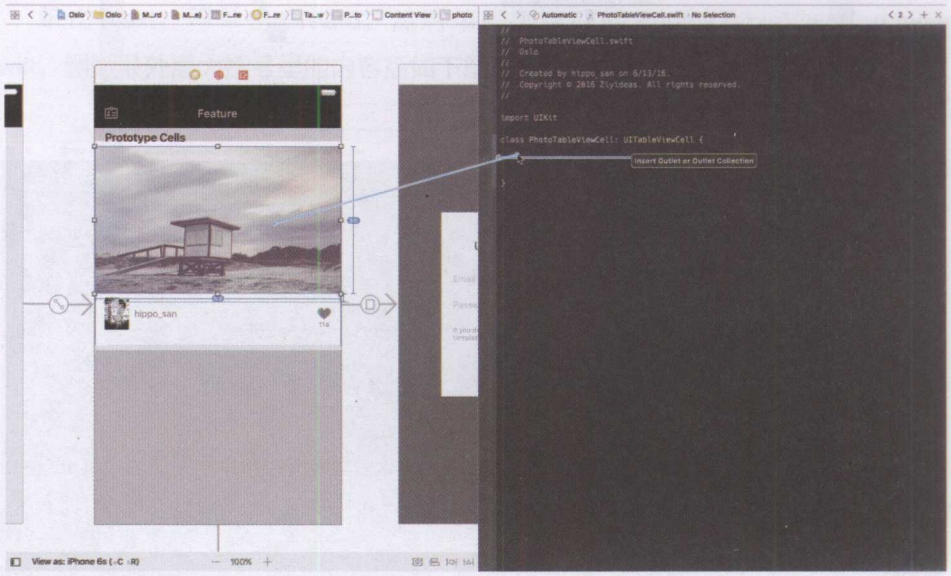
下面把 Storyboard 中的组件在代码中体现，就是用代码来控制这些组件，比如更新界面状态、控制动画等。

点击打开 Assistant Editor。

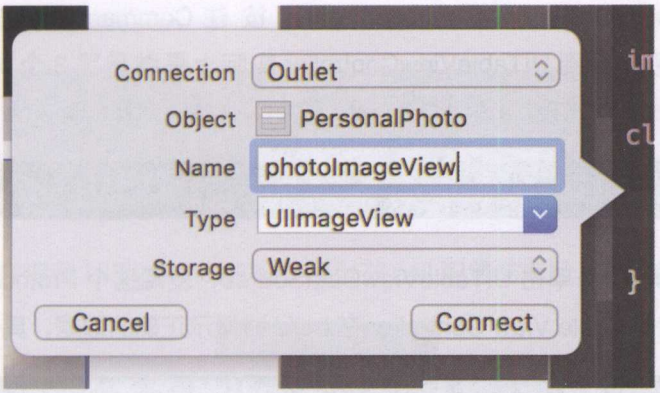


进入 PhotoTableViewCell 的代码控制，如下图所示。

单击 Storyboard 中的 photo，按住 Ctrl 键的同时将其拖曳到 Assistant Editor 中。



在弹出的小窗口的 Name 中填入 photoImageView（习惯组件命名后面加上组件名称）。



单击 Connect，这个过程就将代码和界面组件联系了起来。

以同样的方法将用户头像、用户名、红心图标和下方数字关联相应代码，如果组件太小，可以在 Document Outline 中拖曳放大。最终如下图所示。

```
import UIKit

class PhotoTableViewCell: UITableViewCell {

  @IBOutlet weak var photoImageView: UIImageView!
  @IBOutlet weak var userImageView: UIImageView!
  @IBOutlet weak var userLabel: UILabel!
  @IBOutlet weak var heartImageView: UIImageView!
  @IBOutlet weak var heartCountLabel: UILabel!


}
```

这样 cell 里面的组件就和代码关联完成。之后就可以在 PhotosTableView Controller 中用代码控制界面了。

- Protocol -

Protocol 可以看作是既定的一些方法，如果你接触过 C 语言，可以把它看作是 interface。对于 Photos 这个界面来说，由于使用的是 Table View，所以可以使用 iOS API 中定义行数、自定义 cell 等的 Protocol，能够很方便地在 Table View 中展示内容。

回到 PhotosTableViewController.swift。按住 Command 单击 UITableViewController，可以看到 UITableViewController 实际上是继承了 3 个 class，分别是 UIViewController、UITableViewDelegate 和 UITableViewDataSource：

```
@available(iOS 2.0, *)
open class UITableViewController : UIViewController, UITableViewDelegate, UITableViewDataSource {
```

按住 Command 单击 UITableViewDataSource，发现这个 Protocol 必须使用两个方法，才能使 Table View Controller 在 build 时显示正确的数据，具体方法如下：

```
@available(iOS 2.0, *)
public func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int


// Row display. Implementers should *always* try to reuse cells by setting each cell's reuseIdentifier and querying for available reusable cells with
  dequeueReusableCellWithIdentifier:
// Cell gets various attributes set automatically based on table (separators) and data source (accessory views, editing controls)

@available(iOS 2.0, *)
public func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
```

在 class 内添加如下代码：

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

}

override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

}
```

这时会看到两个红色的错误提示，因为这两个方法都要返回值，可是当前还没有设定这个返回值。先设定 tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int 的返回值，这个方法的意思是在 Table View 里一共包含多少 cell。这里 cell 的数量就是数据本身的数量，即这里暂时返回 1：

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
  return 1
}
```

tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell 这个方法的意思是给每一个 cell 赋予数据。在内部加入如下代码：

```
let cell = tableView.dequeueReusableCell(withIdentifier: "PersonalPhoto") as! PhotoTableViewCell
```

从 Table View 中获得 cell，以便之后对其进行配置。这里因为自定义了 cell 的 class 为 PhotoTableViewCell，所以需要用关键词 cast 让 Controller 知道这个 cell 的类型。既然知道了 cell 是 PhotoTableViewCell，就可以用 PhotoTableViewCell 中

的信息，把数据赋值过去：

```
cell.photoImageView.image = UIImage(named: "photo")
cell.userImageView.image = UIImage(named: "hippo_san")
cell.userLabel.text = "hippo_san"
cell.heartImageView.image = UIImage(named: "heart")
cell.heartCountLabel.text = "114"
```

因为是要求返回 UITableViewCell，所以在 cell 配置结束后，返回就可以了：

```
return cell
```

这样数据部分就完成了，并定义了有多少个 cell，以及这个 cell 内组件的相应配置。

在 iOS 中有很多现成的 Protocol 供我们自定义组件和事件，可以在之后的开发中慢慢接触。

- 自适应高度 -

要想让 Table View 根据 cell 里面的内容自动适应高度，还需要做一些补充。

在 class 中输入 viewd 后 Xcode 会自动补全为 viewDidLoad，按回车键。viewDidLoad 的作用是当视图加载完成时，要做什么信息处理。在方法内先加载父级：

```
super.viewDidLoad()
```

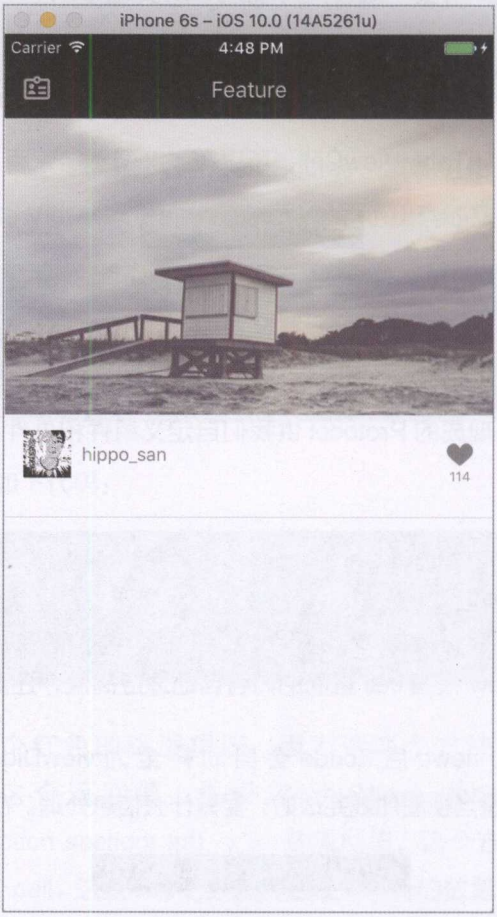
之后编辑 Controller，Table View 的行高是自适应的：

```
tableView.estimatedRowHeight = 303
tableView.rowHeight = UITableViewAutomaticDimension
```

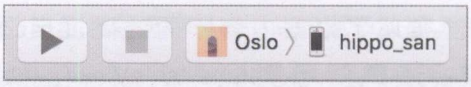
这样 Table View 便实现了根据内容不同的高度自适应。为了定义 Auto Layout 中 cell 的行高，还需要用到 func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat 这个 API，在 class 内部最后添加：

```
override func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat {
  return tableView.frame.width * 0.7
}
```

赶快 Build 一下看看效果吧！在模拟器上出现了之前在 Preview 中看到的设计，如下图所示。



如在模拟器上可以成功运行就代表在实际设备上也能真实运行，由于已经完成了 Auto Layout，所以使用什么设备都应该没问题。将 iOS 设备与电脑相连，在 Xcode 左上角选择设备。



在设备解除锁屏后再次 Build，Oslo 像从 App Store 下载好一样在 Home 出现，而且进入后的界面和模拟器中一样。

如单击 cell 后发现一直是 highlight 状态，就顺手修复吧。在 viewDidLoad 下

方输入 didse，会自动匹配 tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath)，第二个是 override func，选择它，然后输入：

```
tableView.deselectRow(at: indexPath, animated: true)
```

顾名思义，方法本身是用来定义选中了 cell 后要触发的行为，上面的代码则是让被选中的 cell 立刻解除选中。

再次 Build 一下，单击 cell，highlight 消失了。

- Collection View -

Collection View 和 Table View 意思是一样的，所以我们趁热打铁，看看怎么让 Collection View 显示数据信息。

新建 ProfileViewController.swift，删除 class 里面的全部内容：

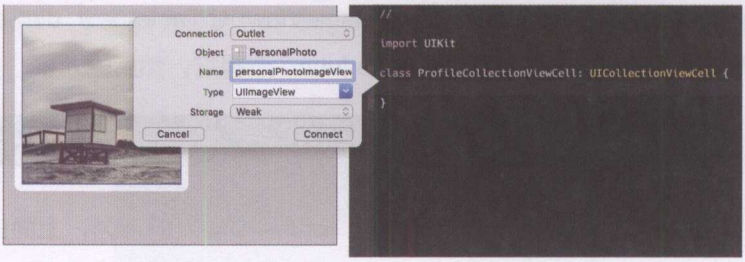
```
import UIKit

class ProfileViewController: UIViewController {



}
```

在 Storyboard 中将 Profile 界面的 class 关联为 ProfileViewController。关联组件到代码中：

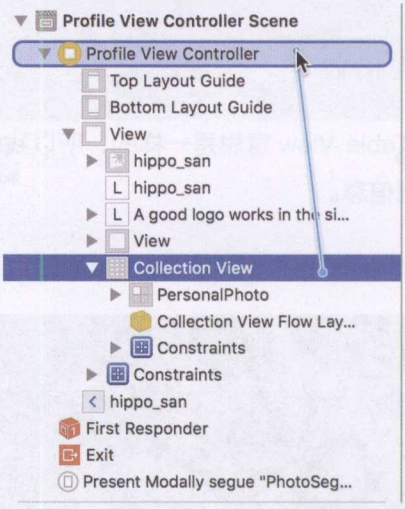
```
@IBOutlet weak var avatarImageView: UIImageView!
@IBOutlet weak var userLabel: UILabel!
@IBOutlet weak var bioLabel: UILabel!
@IBOutlet weak var locationLabel: UILabel!
@IBOutlet var portfolioImage: UIImageView!
@IBOutlet var portfolioName: UIButton!
@IBOutlet var collectionView: UICollectionView!
```

新建 ProfileCollectionViewCell.swift，删除 class 里面的全部内容，在 Storyboard 中关联 cell 的 class 为 ProfileCollectionViewCell。

进入 ProfileCollectionViewCell.swift。打开 Assistant Editor，按住 Ctrl 键拖曳图片。



进入 Main.storyboard，选中 Collection View，按住 Ctrl 键拖曳到 Profile View Controller 上，选中 data source 和 delegate。



同 Table View 与 Table View Controller 的关系不一样，这里的 Collection View 是自定义建立的，View Controller 没有它的任何信息，所以需要手动连接。

进入 ProfileViewController.swift。在 class 外部下方建立如下所示的 extension：

```
extension ProfileViewController: UICollectionViewDataSource {

}
```

写在 extension 里的好处是将 Collection View 的数据和 cell 配置部分从整个 UIViewController 中分离出来，使代码逻辑更清晰。

类似于 UITableViewDataSource，UICollectionViewDataSource 的 protocol

需要执行两个方法：func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int 和 public func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell。这两个方法和 UITableViewDataSource 的作用也是完全一样的。在 extension 中添加如下代码：

```
func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
  return 1
}

func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
  let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "PersonalPhoto", for: indexPath) as! ProfileCollectionViewCell

  return cell
}
```

单击用户头像后看到 Profile 显示正常。

为了定义 cell 的 Auto Layout，这里还需要在文件最后加入以下代码：

```
extension ProfileViewController: UICollectionViewDelegateFlowLayout {
  func collectionView(_ collectionView: UICollectionView, layout collectionViewLayout: UICollectionViewLayout, sizeForItemAt indexPath: IndexPath) -> CGSize {
    return CGSize(width: collectionView.frame.size.height, height: collectionView.frame.size.height)
  }
}
```

- 定义组件事件 -

如果在原型部分 Segue 设置没问题，Build 后单击 navigation bar 左边的 user 图标时会出现 Login 的界面。这个界面相对简单，目前需要解决的问题是单击背景层后 Login 界面消失，返回 Photos 界面。

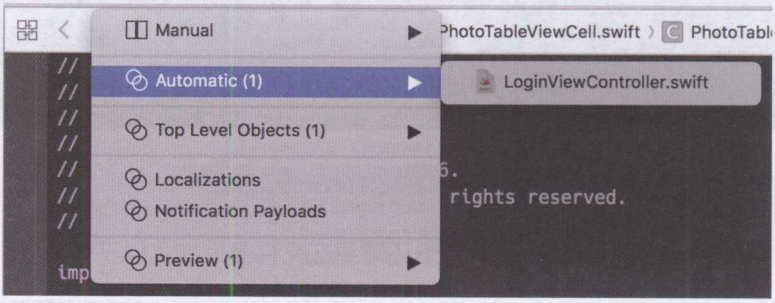
Button 这种特殊组件除了可以连接到代码文件中作为 outlet 外，还可以定义其他事件。

首先连接代码和界面。老办法，在 Xcode 中依次单击 New File... - iOS - Source - Cocoa Touch Class，Subclass of 填 UIViewController，Class 填 LoginViewController，创建。删除 class 内部全部代码：

```
import UIKit

class LoginViewController: UIViewController {



}
```

然后关联代码与界面。在 Main.storyboard 中单击 Login View Controller，在 Identity inspector - Custom Class - Class 中填入 LoginViewController。这样代码和界面就完成了关联。

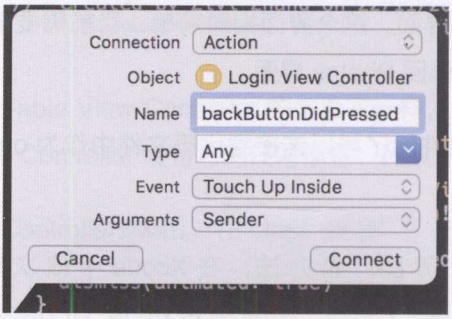
打开 Assistant Editor，因为已经关联了代码和界面，所以在顶部导航选择 Automatic 就能自动进入对应的 LoginViewController 中。



使用之前的 Ctrl 拖曳方法，关联组件和代码，如下：

```
@IBOutlet var loginWebView: UIWebView!
@IBOutlet var backButton: UIButton!
```

在 Storyboard 中选中 button，再次按住 Ctrl 键拖曳到代码中，如下图所示设置。



需要注意的是，在代码中定义按钮样式时，Connection 用 Outlet，定义代码行为时用 Action。

在 func backButtonDidPressed(_ sender: Any) 中加入以下代码：

```
dismiss(animated: true, completion: nil)
```

意思是让当前的 View Controller 消失。

Build 后单击 Login 界面消失，并且返回到了 Photos 界面。

在 Storyboard 中的关联事件

在 Storyboard 的 View Controller 组件中可以看到有 Exit：

但是这时无法连接到 Exit。需要在代码中建立一个方法，比如：

```
@IBAction func dismiss(_ sender: UIStoryboardSegue) {
  dismiss(animated: true, completion: nil)
}
```

然后在 Storyboard 中按住 Ctrl 键拖曳 button 到 Exit，选择 dismiss: 就可以让当前的 View Controller 消失了。

- 触发 Segue -

之前在 Storyboard 中通过 Segue 连接了各个界面，可是如何触发这些 Segue 来达到界面间的过渡呢？比如在 Profile 界面中，单击 Collection View 中的图片 Photo 页面，触发 PhotoSegue。

打开 ProfileViewController.swift，在 extension ProfileViewController: UICollectionViewDataSource, UICollectionViewDelegate 中添加以下代码：

```
func collectionView(_ collectionView: UICollectionView, didSelectItemAt indexPath: IndexPath) {
  performSegue(withIdentifier: "PhotoSegue", sender: self)
}
```

Build 一下单击图片就跳转到 Photo 界面了。

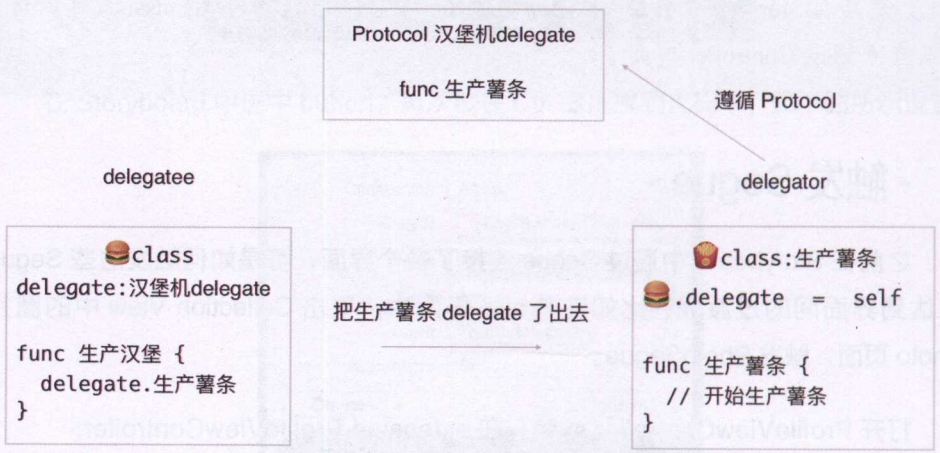
至于 Photos Table View Controller 界面的图片，其实是在 Photos Table View

Cell 上的，怎样才能单击 Cell 上的图片，并调用 View Controller 的 performSegue 方法？这就要用到 Delegate 了。

- Delegate -

Delegate 是 iOS 开发中最常见的程序设计模式，iOS 本身的 API 也在大量使用这个结构。Delegate 这种设计模式对一些新近的开发者来说理解起来可能会有点绕，下面先来看一个例子。

有一台汉堡机和一台薯条机，在卖套餐的时候，需要汉堡机生产出来一个汉堡时，薯条机就要自动生产出一包薯条。因为汉堡机无法生产薯条，就需要把生产薯条这个任务 Delegate（委任）给薯条机，这时薯条机就是 Delegator，汉堡机就是 Delegatee。那么如何在汉堡生产的时候让薯条机知道要开始生产薯条了呢？这就需要通过 Protocol 来调用对应的任务。



设置 Delegate 步骤如下：

- 定义 Protocol，把需要 Delegate 的方法在其中定义好。
- 在 Delegatee 中定义变量 delegate。

- 在 Delegatee 中把需要 Delegator 执行的方法写到对应的地方。
- 让 Delegator 遵循 Protocol。
- 在 Delegator 中设置 Delegatee 的 Delegate 为自身（这样才能正确地关联汉堡机和薯条机）。
- 在 Delegator 中定义 Protocol 中的方法。

回到 Oslo 中，想实现单击 cell 中不同的内容到不同的页面中，需要用到 performSegue 这个在 View Controller 中的方法，因此这里 Cell 是 Delegatee，Table View Controller 是 Delegator，每次单击 cell，View Controller 就会运行 PerformSegue。

按照上面的 Delegate 设置步骤，先在 PhotoTableViewCell.swift 中，class 外部定义 protocol：

```
protocol PhotoTableViewCellDelegate: class {
  func tapToPerformSegue(_ sender: Any)
}
```

在 class 内部定义变量 delegate：

```
weak var delegate: PhotoTableViewCellDelegate?
```

在 class 内部最下方输入以下代码：

```
func tapped(_ sender: Any) {
  if let tag = (sender as AnyObject).view?.tag {
    switch tag {
    case 0:
      delegate?.tapToPerformSegue(sender)

    case 1:
      delegate?.tapToPerformSegue(sender)

    default:
      break
    }
  }
}
```

这里定义了 tapped 方法，在单击图片或者用户头像时会调用这个方法。如果组件的 tag 是 0 或者 1，则会调用 tapToPerformSegue 这个方法。

切换到 Main.storyboard 中，单击 Photo Image View，将 Attribute inspector - View - Tag 设置为 0。同样的方法将 User Image View 的 Tag 设置为 1。

iOS 中 Image View 是没有 action 的，因此要在 Image View 上增加点击手势来实现点击。设置 photoImageView 和 userImageView 的 didSet：

```
@IBOutlet weak var photoImageView: UIImageView! {
  didSet {
    if photoImageView.gestureRecognizers == nil {
      photoImageView.addGestureRecognizer(UITapGestureRecognizer(target: self, action: #selector(tapped)))
    }
  }
}

@IBOutlet weak var userImageView: UIImageView! {
  didSet {
    if userImageView.gestureRecognizers == nil {
      userImageView.addGestureRecognizer(UITapGestureRecognizer(target: self, action: #selector(tapped)))
    }
  }
}
```

每次手势单击两个 Image View 时，都会触发 tapped 方法。

切换到 PhotosTableViewController，在文件最下方加入 extension，让 class 遵循 Protocol，并定义 Protocol 中的方法：

```
extension PhotosTableViewController: PhotoTableViewCellDelegate {
  func tapToPerformSegue(_ sender: Any) {
    if let tag = (sender as AnyObject).view?.tag {
      switch tag {
      case 0:
        performSegue(withIdentifier: "PhotoSegue", sender: sender)

      case 1:
        performSegue(withIdentifier: "ProfileSegue", sender: sender)

      default:
        break
      }
    }
  }
}
```

在 tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell 中设置 Cell 的 delegate 为自身：

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
  let cell = tableView.dequeueReusableCell(withIdentifier: "PersonalPhoto") as! PhotoTableViewCell
  cell.delegate = self

  return cell
}
```

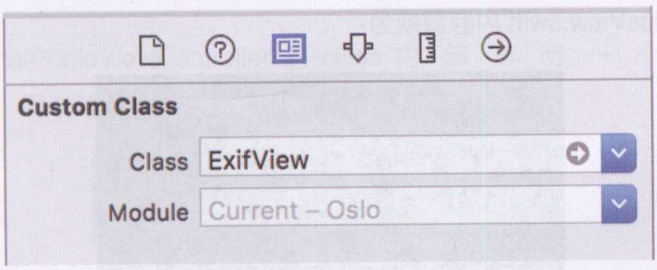
Build 一下，单击 Cell 的图片和用户信息部分都能到对应的页面了。

> Delegate vs. NotificationCenter
>
> Delegate 和 NotificationCenter 一样，都是发出“通知”，让定义的地方执行相应方法。区别在于 Delegate 是一对一的，NotificationCenter 是一对多的。如果需要一个地方的触发要通知到多个地方触发不同的方法，就应该用 NotificationCenter。

- xib-

之前已经创建好的 xib 界面，如何用在 View Controller 中呢？其实很简单，只需要加载 xib 后再添加到现有的 View 上即可。下面我们就来实现单击 statistics 和 camera 后出现对应的 xib。

首先将 xib 文件关联到对应的代码文件中，创建 ExifView.swift 和 StatisticsView.swift 两个文件。并将它们分别关联到对应的 xib 上，关联方式和 View Controller 相同。



关联 xib 组件和代码。将 ExifView.swift 内容替换为：

```
import UIKit

class ExifView: UIView {
  @IBOutlet var createdTimeTitleLabel: UILabel! {
    didSet {
      createdTimeTitleLabel.text = "Published"
    }
  }
  @IBOutlet weak var createdTimeLabel: UILabel!
  @IBOutlet var dimensionTitleLabel: UILabel! {
    didSet {
      dimensionTitleLabel.text = "Dimensions"
    }
  }
  @IBOutlet weak var dimensionsLabel: UILabel!
  @IBOutlet var makeTitleLabel: UILabel! {
    didSet {
      makeTitleLabel.text = "Camera Make"
    }
  }
  @IBOutlet weak var makeLabel: UILabel!
  @IBOutlet var modelTitleLabel: UILabel! {
    didSet {
      modelTitleLabel.text = "Camera Model"
    }
  }
  @IBOutlet weak var modelLabel: UILabel!
  @IBOutlet var apertureTitleLabel: UILabel! {
    didSet {
      apertureTitleLabel.text = "Aperture"
    }
  }
  @IBOutlet weak var apertureLabel: UILabel!
  @IBOutlet var exposureTitleLabel: UILabel! {
    didSet {
      exposureTitleLabel.text = "Exposure Time"
    }
  }
  @IBOutlet weak var exposureTimeLabel: UILabel!
  @IBOutlet var focalTitleLabel: UILabel! {
    didSet {
      focalTitleLabel.text = "Focal Length"
    }
  }
  @IBOutlet weak var focalLengthLabel: UILabel!
  @IBOutlet var isoTitleLabel: UILabel! {
    didSet {
      isoTitleLabel.text = "ISO"
    }
  }
  @IBOutlet weak var isoLabel: UILabel!
}
```

将 StatisticsView.swift 内容替换为：

```
import UIKit

class StatisticsView: UIView {
  @IBOutlet var downloadsTitleLabel: UILabel! {
    didSet {
      downloadsTitleLabel.text = "Downloads"
    }
  }
  @IBOutlet weak var downloadsLabel: UILabel!
  @IBOutlet var viewsTitleLabel: UILabel! {
    didSet {
      viewsTitleLabel.text = "Views"
    }
  }
  @IBOutlet weak var viewsLabel: UILabel!
  @IBOutlet var likesTitleLabel: UILabel! {
    didSet {
      likesTitleLabel.text = "Likes"
    }
  }
  @IBOutlet var likesLabel: UILabel!
}
```

创建 Misc.swift 文件，将内容替换为：

```
public extension UIView {
  public class func load(from xib: String, with frame: CGRect) -> UIView? {
    guard let nibView = Bundle.main.loadNibNamed(xib, owner: self, options: nil) as? [UIView] else { return nil }
    let view = nibView[0]
    view.frame = frame
    view.autoresizingMask = [.flexibleWidth, .flexibleHeight]

    return view
  }
}
```

这里定义了如何在 UIView 上加载一个 xib。从 Bundle 中读取 xib 文件，然后设置了大小。

创建 PersonalPhotoViewController.swift 文件，关联到 Photo View Controller 上，然后连接组件和代码：

```
@IBOutlet var personalPhotoImageView: UIImageView!
@IBOutlet var heartButton: UIButton!
@IBOutlet var heartCountLabel: UILabel!
@IBOutlet var downloadButton: UIButton!
@IBOutlet var shareButton: UIButton!
@IBOutlet var statisticsButton: UIButton!
@IBOutlet var exifButton: UIButton!
@IBOutlet var savePhotoLabel: UILabel!
```

定义背景按钮事件：

```
@IBAction func closeButtonPressed(_ sender: Any) {
  dismiss(animated: true)
}
```

在 PersonalPhotoViewController 的 class 内部声明一个 exifView 和 statisticsView，用来储存加载的 xib：

```
var exifView: UIView?
var statisticsView: UIView?
```

定义 statistics 点击事件，但要注意和组件相连接：

```
@IBAction func statisticsButtonDidPressed(_ sender: Any) {
  if statisticsView == nil {
    if exifView != nil {
      exifButton.setBackgroundImage(camera, for: .normal)

      exifView?.removeFromSuperview()
      exifView = nil
    }

    statisticsButton.setBackgroundImage(  statistics-on, for: .normal)

    if let view = UIView.load(from: "StatisticsView", with: personalPhotoImageView.bounds)  as? StatisticsView {
      statisticsView = view

      personalPhotoImageView.addSubview(statisticsView!)
    }
  } else {
    statisticsButton.setBackgroundImage(statistics, for: .normal)

    statisticsView?.removeFromSuperview()
    statisticsView = nil
  }
}
```

同样定义 camera 单击事件：

```
@IBAction func exifButtonDidPressed(_ sender: Any) {
  if exifView == nil {
    if statisticsView != nil {
      statisticsButton.setBackgroundImage(statistics, for: .normal)

      statisticsView?.removeFromSuperview()
      statisticsView = nil
    }

    exifButton.setBackgroundImage(  camera-on, for: .normal)

    if let view = UIView.load(from: "ExifView", with: personalPhotoImageView.bounds) as? ExifView {
      exifView = view

      personalPhotoImageView.addSubview(exifView!)
    }
  } else {
    exifButton.setBackgroundImage(camera, for: .normal)

    exifView?.removeFromSuperview()
    exifView = nil
  }
}
```

如果 exifView 还不存在，就需要加载一个并添加到 Photo 上；如果已经存在就将其移除。statisticsView 同理。注意这里 removeFromSuperview 只是将 View 在显示上移除了，但是 exifView 还是有赋值的，并不等于 nil。

Build 并单击 statistics 按钮，至此就完成了。

- App Security -

App Security 是 iOS 10 上新加入的一个要求。如果应用要访问用户的敏感数据，或者提供跟安全相关的操作，需要通知用户，并提供相应的说明，这就是 App Security 的作用。Oslo 会保存图片到用户 Photos 应用的相册中，确认访问了用户的照片，所以需要用到 App Security。

在 class 内部添加如下代码，要注意和组件相连接：

```
@IBAction func downloadButtonDidPressed(_ sender: Any) {
  guard personalPhoto != nil else { return }
  UIImageWriteToSavedPhotosAlbum(personalPhoto!, self, #selector(save(_:didFinishSavingWithError:contextInfo:)), nil)
}
```

UIImageWriteToSavedPhotosAlbum 这个 API 即为保存图片到相册的 API。按住 Command 单击后发现 Swift 化还不够形象，希望日后能有所改善。这时候会报错，因为 save 方法还没有定义。现在就在 class 内部最下方加入如下代码定义：

```
@objc private func save(_ image: UIImage, didFinishSavingWithError error: NSError?, contextInfo: UnsafeRawPointer) {
  saveLabel.isHidden = false

  if let error = error {
    saveLabel.text = "⚠️ Photo is not saved. You may check the photo permission or storage."
    print(error.localizedDescription)
  } else {
    saveLabel.text = "🖼 Photo is saved to Album."
  }

  delay(2.0) {
    self.saveLabel.isHidden = true
  }
}
```

注意这里的 @objc，是因为 selector 使用 private 的方法暴露给 Objective-C，其他部分就很好理解了。提示会在两秒钟后自动消失。Build 一下，单击保存按钮发现没有反映，在 debug 信息中发现这样的描述：

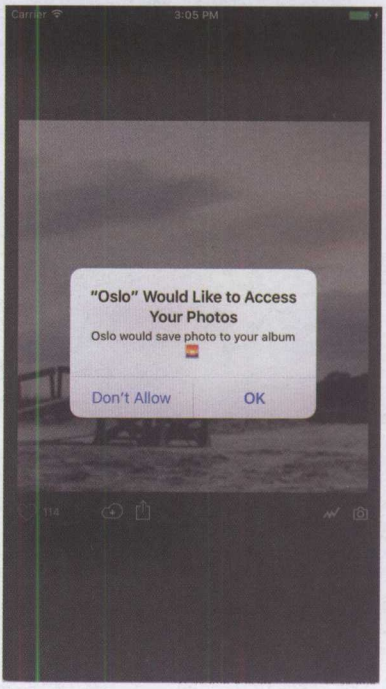
```
2016-08-17 14:59:16.155830 Oslo[32702:2304711] [access] This app has crashed because it attempted to access privacy-sensitive data without a usage description.
The app's Info.plist must contain an NSPhotoLibraryUsageDescription key with a string value explaining to the user how the app uses this data.
```

说明应用访问到了隐私数据，需要提供一个说明给用户。打开 Info.plist，单击 Information Property List 旁边的“+”号，在 Key 中填写 Privacy - Photo Library Usage Description[1]，在 Value 中填写一个向用户说明的理由，比如 Oslo would save photo to your album：

Information Property List	Dictionary	(16 items)
Privacy - Photo Library Usage Des...	String	Oslo would save photo to your album
Localization native development re...	String	en
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
▶ Required device capabilities	Array	(1 item)
▶ Supported interface orientations	Array	(3 items)
▶ Supported interface orientations (i...	Array	(4 items)

1 完整的 Privacy 列表可以在 https://developer.apple.com/library/prerelease/content/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html 查看。

再次 Build，单击保存图片后会出现提示。



单击“OK”后就可以正常保存。

想让这里更有趣一点的话，就让 emoji 随机显示吧（这是可选的，做不做完全由你）。进入 Misc.swift，我们来定义一个随机访问数组角标的方法。在文件最后加入：

```
extension Array {
  func randomItem() -> Element {
    let index = Int(arc4random_uniform(UInt32(self.count)))
    return self[index]
  }
}
```

这里定义了一个区间为 0 到数组个数上限的 index 随机数，然后作为角标返回了对应的元素。

回到 PersonalPhotoViewController.swift，在 IBOutlet 上方添加：

```
private let emoji: Array = ["🌄", "🌅", "🌠", "🎇", "🎆", "🌇", "🌆", "🏙", "🌃", "🌌", "🌉", "🌁"]
```

这里的 emoji 也可以自由添加。然后将 save 方法中的：

```
"🌟 Photo is saved to Album."
```

改为：

```
"\(emoji.randomItem())" + " Photo is saved to Album."
```

Build 一下，多单击几次保存按钮，emoji 每次变化都不一样。

- UIActivity -

UIActivity 是一个通过传输数据来调用服务的 class，现在就用它来实现 Oslo 的图片分享功能。

在 PersonalPhotoViewController.swift 中，在 class 内加入如下代码，注意要和组件相连接：

```
@IBAction func shareButtonDIdPressed(_ sender: Any) {
  let shareImage: UIImage = personalPhotoImageView.image!
  let activityItems = [shareImage] as [Any]
  let activityViewController = UIActivityViewController(activityItems: activityItems, applicationActivities: nil)

  if UIDevice.current.userInterfaceIdiom == .pad {
    activityViewController.popoverPresentationController?.sourceView = self.view
  }

  self.present(activityViewController, animated: true)
}
```

这段代码中的 activityItems 是关键，它包含了要传输的数据，可以是 String 或 URL，也可以是图片等，将这些赋值到一个 Array 中，通过 UIActivityViewController 传输给对应的其他服务。

这里添加了针对于 iPad 的代码，iPad 的分享会由一个 Pop View 来完成。

注意，不同的服务对传输数据的要求也是不一样的。比如，如果只传 String 和图片，则无法分享到微信。

Build 一下试试分享功能吧。

- @IBInspectable -

现在界面和设计图不一样的地方就是组件本身的样式问题了，比如圆角、边框、

阴影等。在 iOS 中有一个概念是 CALayer，有点类似于在 Sketch 中的 layer，负责绘图和成像。也许你会问这不是 UIView 做的事情吗？这样理解没错，不过 CALayer 更多的是负责一些复杂的动画和渲染等工作。其实 UIView 更多的是为 CocoaTouch 设备设计的，比如 iPhone/iPad，直接使用一些手势之类的操作。而 CALayer 是多个平台都可以共用的，比如想开发一款 iOS、macOS、tvOS 同时兼容的图表曲线展示的应用，那绘制在 CALayer 是最合适不过了。

明白这个概念后，就可以使用 CALayer 先实现组件的自定义展示。Storyboard 还不支持 CALayer 的直接操作，但是界面的样式工作放在代码里又显得多余，这个时候就要用到 @IBInspectable。@IBInspectable 可以用来定义一些 runtime 的样式属性，因为是 inspectable，所以也不用在代码中设置，而是在 Attribute inspector 中就可以完成。

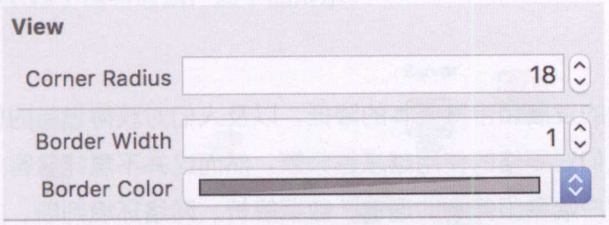
在 Misc.swift 最后添加如下代码：

```
extension UIView {
  @IBInspectable var cornerRadius: CGFloat {
    get {
      return layer.cornerRadius
    }
    set {
      layer.cornerRadius = newValue
      layer.masksToBounds = newValue > 0
    }
  }

  @IBInspectable var borderWidth: CGFloat {
    get {
      return layer.borderWidth
    }
    set {
      layer.borderWidth = newValue
    }
  }

  @IBInspectable var borderColor: UIColor? {
    get {
      return UIColor(cgColor: layer.borderColor!)
    }
    set {
      layer.borderColor = newValue?.cgColor
    }
  }
}
```

这里定义了设计中涉及的圆角、边框宽度和边框颜色。

我们先来修改 Photos Table View Cell 中的用户头像。打开 Main.storyboard，选中 User Image View，会发现 Attribute inspector 中多了一个 View 的区域，可以

可视化调节代码控制的属性。按设计图设置如下：



Build 一下，这样就实现了圆角和边框。

使用同样的方法，将 Profile View Controller 中的用户头像也设置好，同时为 Collection View 中的 Image View 设置好圆角。

需要注意的是，如果 Collection View 中的 Image View 既想显示圆角，又想显示阴影，则需要将阴影加在 cell 上。进入 ProfileCollectionViewCell.swift，在 class 内部最后添加：

```
override func layoutSubviews() {
  super.layoutSubviews()

  self.layer.shadowColor = UIColor.black.cgColor
  self.layer.shadowOpacity = 0.1
  self.layer.shadowRadius = 3.0
  self.layer.shadowOffset = CGSize(width: 2, height: 4)
}
```

然后在 Main.storyboard 中单击 Collection View 中的 Image View，勾选 Attribute inspector - View - Clip To Bounds，这样可以让圆角生效。注意，不要勾选 cell 的 Clip To Bounds。如果圆角和阴影同时加在 Image View 上，会导致让圆角生效的 Clip To Bounds 阴影被剪（clip）掉。

网络

随着云技术的发展和带宽成本的降低，以及人们对获得最新的信息和他人互动的渴望，使得人们对网络的使用越来越频繁。然而这并不意味着客户端的工作越来越少，相反，客户端和服务端的通信、数据解析、网络环境判断、同步异步处理、异常分析等技术，加大了客户端的工作量和开发的复杂程度。

由于笔者是从 Web 开发转移到 App 开发的，所以当时在接触 App 网络开发时，有点像 Web 时从静态页面进入了可以登录注册、更新数据的神奇世界中，现在回想起，那种感觉也是油然而生啊！

虽然网络开发十分辛苦，但是取得的成果是可以复用的。比如当再做不同的应用时，请求网络数据和解析数据这些基本代码程序就不用再写一次，并且同客户端本身的开发有很大不同。

可能你还不太清楚网络是怎样和客户端通信的，以及是怎样传输数据的，不用着急，接下来我们一步步了解网络，让应用“活”起来。

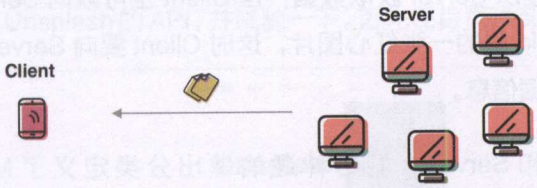
- Client & Server -

在介绍网络数据传输之前，先来了解一下 Client（客户端）和 Server（服务端）。Client 和 Server 不仅仅存在于移动应用中，也同样存在于 Web 应用中。

Client 就是我们使用的手机、电脑、平板等设备，而 Server 也是能看得见摸得着的设备，只是它们一般都在机房，机房通常由专业的托管机构维护，我们接触不到。Client 一般是一台设备，Server 一般是几台机器的集群。

Client 一侧的开发往往被称作 Frontend（前端）开发，Server 一侧的开发，则被称作 Backend（后端）开发。Frontend 负责实现样式、利用设备本身特性开发功能等，也就是之前我们一直在做的事情。Backend 负责数据存储和利用 API 将数据传输给 Client。

可是 Server 如何知道应该传输哪些数据给 Client 呢？这数据的传输又是怎样实现的呢？这些都依赖于网络通信，如下图所示。

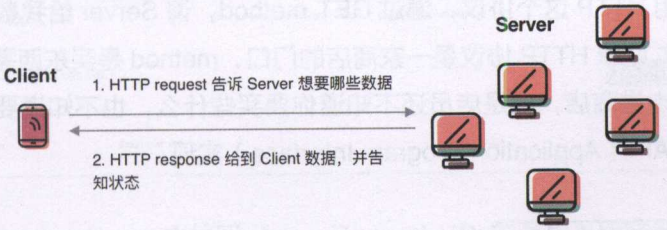


- 通信 -

网络通信理解起来很简单，就好比两个人说话一样，两台机器之间也可以通过通信说话，只不过两人说话的媒介是震动带来的声音，而计算机之间的通信是通过 protocol（协议）。

这个 Protocol 和 Swift 中的 Protocol 有些类似，都是双方遵从一个统一的信息，只要信息相互联接对上，双方就可以自由交谈，这个“对上”，网络中的定义是 handshake（握手）。

那么被广泛使用的网络通信 Protocol 是什么呢？我想你一定听说过 HTTP（HyperText Transfer Protocol）。Client 通过发送 HTTP request 给 Server，Server 根据接收到 request，返回数据给 Client，这个返回的数据叫 response。由于 response 有多种情况，比如没有找到数据或延迟时间过长等，所以又多出一个概念 status code[1]，用来标记 response 的状态，告知 Client。



1 在 https://en.wikipedia.org/wiki/List_of_HTTP_status_codes 可以找到完整的 HTTP status codes。

- HTTP Request Methods -

之前介绍的只是从 Server 获取数据，但 Client 也可以向 Server 发送数据甚至修改数据。比如 Oslo 里的一张红心图片，这时 Client 要向 Server 发送数据，来更新 Server 的红心数据信息。

为了对 Client 和 Server 之间各种通信做出分类定义了 Methods。不同的 Methods 代表了不同的通信方式[1]。这里只列出最常用的几个[2]。

Method	作用（遵从 REST 原则）
GET	仅从 Server 获取数据
POST	创建或更新 Server 数据
PUT	创建或更新 Server 数据，和 POST 的区别是 PUT 具有 idempotent（幂等）
DELETE	删除 Server 数据

请务必弄清以上几个 Methods 具体的区别及在什么场合下使用，因为这几个概念非常重要。

就 Oslo 这个具体应用来说，在 Unsplash 中，对几个 methods 进行了明确的规定，可以在 https://unsplash.com/documentation#http-verbs 查看。

- API -

现在我们知道了 Client 和 Server 是如何通信的，并且用了什么方法，比如 Client 说“我用 HTTP 这个协议，通过 GET method，请 Server 给我数据”，可是给什么数据呢？好像 HTTP 协议是一家商店的门口，method 是买东西需要的钱，你拿着钱从门口走进商店，但是店员还不知道你要买些什么，也不知道要给你什么。这时就要通过 API（Application Program Interface）实现。

---

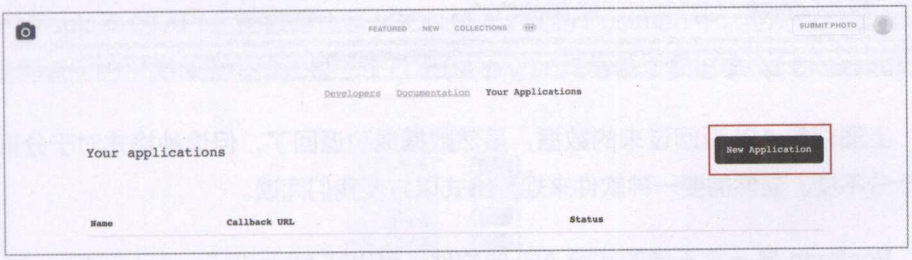
1 这里按照 REST https://en.wikipedia.org/wiki/Representational_state_transfer 原则来列举 methods 的作用。

2 https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html 可以查看完整的 HTTP request methods。

在网络请求中，API 可以被简单地理解为 URL。比如 https://unsplash.com/new，这其实就是一个 API，打开后看到的图片就是 API 返回来的数据，很容易吧！

下面了解一下 Unsplash 的 API，并试验一下。首先在首页顶部位置进入 API 文档。

单击 Register as a developer 注册一个开发者账户（开发者，这个身份听起来不错）。注册后会显示 API 的守则，阅读没问题后单击 Continue。在 What do you want to build? 中填写想用 API 做的事，勾选同意选项后单击 Accept。如果想在自己的应用中使用 Unsplash API，需要在 Unsplash 中建立一个自己的应用，单击页面右方的 New Application。



在 Application Name 中填写 Oslo。Redirect URI 的意思是跳转地址，比如在用户登录的时候，会出现一个 Unsplash 的授权页面，授权成功后，授权页面消失，并跳转到另一个页面，这里填写 oslo://photos。勾选下方所有选项后确定，这样 Oslo 的应用就注册完成！ Application ID 和 Secret 是用来请求 API 时传的参数，让

Unsplash 知道是从 Oslo 发过来的请求。

在发送 API 请求以前，先在 https://unsplash.com/documentation 中了解 Unsplash 的 API。这里提供了非常多的 API 来获取各种类型不同的数据，也提供了很多 response 的实例。为了体验什么是 API，我们先用 GET 来做个试验。

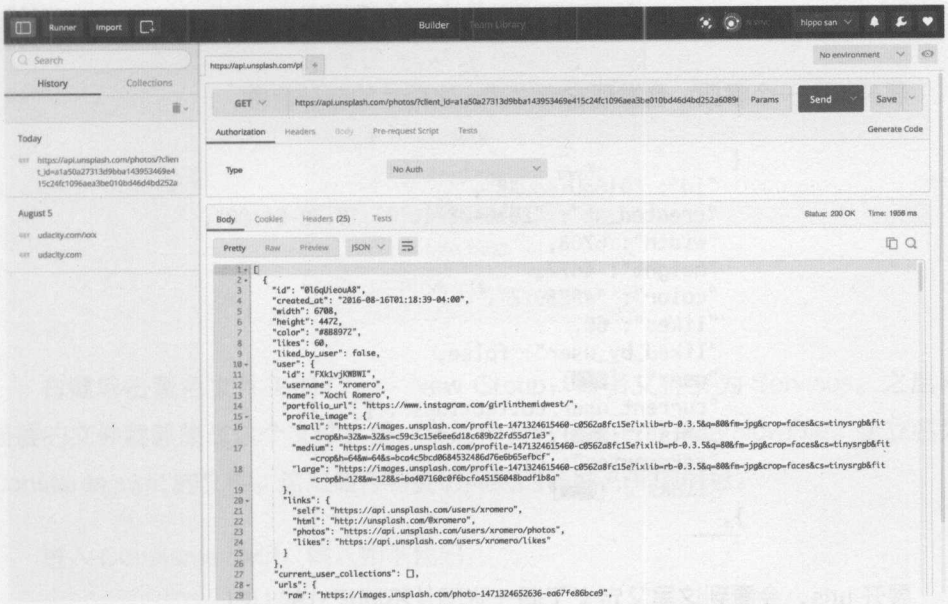
https://unsplash.com/documentation#authorization 举例了通过验证获得图片的 API，也就是 https://api.unsplash.com/photos/?client_id=YOUR_APPLICATION_ID，其中 YOUR_APPLICATION_ID 为刚刚注册后获得的 Application ID，在访问这个 API 时要将 YOUR_APPLICATION_ID 替换为 Application ID。

既然 API 是 URL，那么就可以在浏览器的 URL 输入框里输入访问，之后会显示如下页面。

[{"id":"0l6qUieouA8","created_at":"2016-08-16T01:18:39-04:00","width":6708,"height":4472,"color":"#888972","likes":60,"liked_by_user":false,"user":{"id":"FXk1vjKWBWI","username":"xromero","name":"Xochi Romero","portfolio_url":"https://www.instagram.com/girlinthemidwest/","profile_image":{"small":"https://images.unsplash.com/profile-1471324615460-c0562a8fc15e?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=faces\u0026cs=tinysrgb\u0026fit=crop\u0026h=32\u0026w=32\u0026s=c59c3c15e6ee6d18c689b22fd55d71e3","medium":"https://images.unsplash.com/profile-1471324615460-c0562a8fc15e?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=faces\u0026cs=tinysrgb\u0026fit=crop\u0026h=64\u0026w=64\u0026s=bca4c5bcd0684532486d76e6b65efbcf","large":"https://images.unsplash.com/profile-1471324615460-c0562a8fc15e?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=faces\u0026cs=tinysrgb\u0026fit=crop\u0026h=128\u0026w=128\u0026s=ba407160c0f6bcfa45156048badf1b8a"},"links":{"self":"https://api.unsplash.com/users/xromero","html":"http://unsplash.com/@xromero","photos":"https://api.unsplash.com/users/xromero/photos","likes":"https://api.unsplash.com/users/xromero/likes"}},"current_user_collections":[],"urls":{"raw":"https://images.unsplash.com/photo-1471324652636-ea67fe86bce9","full":"https://hd.unsplash.com/photo-1471324652636-ea67fe86bce9","regular":"https://images.unsplash.com/photo-1471324652636-ea67fe86bce9?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=entropy\u0026cs=tinysrgb\u0026w=1080\u0026fit=max\u0026s=d789eecdd8266085d887e726c416bd7e","small":"https://images.unsplash.com/photo-1471324652636-ea67fe86bce9?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=entropy\u0026cs=tinysrgb\u0026w=400\u0026fit=max\u0026s=19d43fea7c69a05c85f59bb80c01e1d9","thumb":"https://images.unsplash.com/photo-1471324652636-ea67fe86bce9?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=entropy\u0026cs=tinysrgb\u0026w=200\u0026fit=max\u0026s=7aa8c2e50beccb12c24da4b4c93d7de1"},"categories":[],"links":{"self":"https://api.unsplash.com/photos/0l6qUieouA8","html":"http://unsplash.com/photos/0l6qUieouA8","download":"http://unsplash.com/photos/0l6qUieouA8/download"}},{"id":"7MBgGXQ2Rqg","created_at":"2016-08-15T22:18:49-04:00","width":2304,"height":1536,"color":"#B0B0BD","likes":42,"liked_by_user":false,"user":{"id":"IT5AO4bCfbk","username":"agaputrantara","name":"Aga Putra","portfolio_url":null,"profile_image":{"small":"https://images.unsplash.com/profile-1469525901428-23d9ba2ec5d2?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=faces\u0026cs=tinysrgb\u0026fit=crop\u0026h=32\u0026w=32\u0026s=e2f1e400194af884b5d2678d4a2a1618","medium":"https://images.unsplash.com/profile-1469525901428-23d9ba2ec5d2?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=faces\u0026cs=tinysrgb\u0026fit=crop\u0026h=64\u0026w=64\u0026s=bb7136d8f34ccdec7d9024da78e68af0","large":"https://images.unsplash.com/profile-1469525901428-23d9ba2ec5d2?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=faces\u0026cs=tinysrgb\u0026fit=crop\u0026h=128\u0026w=128\u0026s=ff41692047e002e00d1a9649d3f119eb"},"links":{"self":"https://api.unsplash.com/users/agaputrantara","html":"http://unsplash.com/@agaputrantara","photos":"https://api.unsplash.com/users/agaputrantara/photos","likes":"https://api.unsplash.com/users/agaputrantara/likes"},"current_user_collections":[],"urls":{"raw":"https://images.unsplash.com/photo-1471313902629-21dffc56c966","full":"https://hd.unsplash.com/photo-1471313902629-21dffc56c966","regular":"https://images.unsplash.com/photo-1471313902629-21dffc56c966?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=entropy\u0026cs=tinysrgb\u0026w=1080\u0026fit=max\u0026s=cd3bd472a30ea5c54e9c5cb5576d8cc3","small":"https://images.unsplash.com/photo-1471313902629-21dffc56c966?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=entropy\u0026cs=tinysrgb\u0026w=400\u0026fit=max\u0026s=9414788c558cc56c76888fa11fa5d809","thumb":"https://images.unsplash.com/photo-1471313902629-21dffc56c966?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=entropy\u0026cs=tinysrgb\u0026w=200\u0026fit=max\u0026s=6c86bf967db6cca45a580f5fcc8e9f58"},"categories":[],"links":{"self":"https://api.unsplash.com/photos/7MBgGXQ2Rqg","html":"http://unsplash.com/photos/7MBgGXQ2Rqg","download":"http://unsplash.com/photos/7MBgGXQ2Rqg/download"}},{"id":"OCzolrQ307Q","created_at":"2016-08-15T21:32:36-04:00","width":5815,"height":3866,"color":"#F2E6D8","likes":23,"liked_by_user":false,"user":{"id":"yoc9J5Eq9v8","username":"lyndsburk","name":"Lyndsey B","portfolio_url":"https://www.lightstock.com/users/lyndseyb","profile_image":{"small":"https://images.unsplash.com/profile-1468034357471-6753f22ff177?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=faces\u0026cs=tinysrgb\u0026fit=crop\u0026h=32\u0026w=32\u0026s=4968f1320575265d6c03a620f2a62042","medium":"https://images.unsplash.com/profile-1468034357471-6753f22ff177?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=faces\u0026cs=tinysrgb\u0026fit=crop\u0026h=64\u0026w=64\u0026s=85aefa573f7c86a6d56ae3501195033f","large":"https://images.unsplash.com/profile-1468034357471-6753f22ff177?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=faces\u0026cs=tinysrgb\u0026fit=crop\u0026h=128\u0026w=128\u0026s=f26f9a5e326db92b915c19deb8907107"},"links":{"self":"https://api.unsplash.com/users/lyndsburk","html":"http://unsplash.com/@lyndsburk","photos":"https://api.unsplash.com/users/lyndsburk/photos","likes":"https://api.unsplash.com/users/lyndsburk/likes"},"current_user_collections":[],"urls":{"raw":"https://images.unsplash.com/photo-1471311076146-728d848e36d3","full":"https://hd.unsplash.com/photo-1471311076146-728d848e36d3","regular":"https://images.unsplash.com/photo-1471311076146-728d848e36d3?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=entropy\u0026cs=tinysrgb\u0026w=1080\u0026fit=max\u0026s=595ddde174c2dae663ee2229bde028b9","small":"https://images.unsplash.com/photo-1471311076146-728d848e36d3?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=entropy\u0026cs=tinysrgb\u0026w=400\u0026fit=max\u0026s=c23478790ff87d969afadc88d9900fdc","thumb":"https://images.unsplash.com/photo-1471311076146-728d848e36d3?ixlib=rb-0.3.5\u0026q=80\u0026fm=jpg\u0026crop=entropy\u0026cs=tinysrgb\u0026w=200\u0026fit=max\u0026s=08e207fa99c9dd4b0fa5df0f2d4a92e1"},"categories":[],"links":{"self":"https://api.unsplash.com/photos/OCzolrQ307Q","html":"http://unsplash.com/photos/OCzolrQ307Q","download":"http://unsplash.com/photos/OCzolrQ307Q/download"}},{"id":"0eU6ixlbWYI","created_at":"2016-08-15T21:01:09-04:00","width":3849,"height":2730,"color":"#CECED2","likes":41,"liked_by_user":false,"user":{"id":"RCFLNwHvxQs","username":"dexetrix","name":"Mishael

上图就是 API 返回过来的数据，虽然数据成功返回了，但这种格式对于分析理解十分不便。显然需要一种软件来规范格式以方便我们阅读。

Postman 是一款不错的实验 API 的软件，可以在 https://www.getpostman.com/app/download/osx64 下载，在 Postman 中调用 API 会得到比较直观的结果。

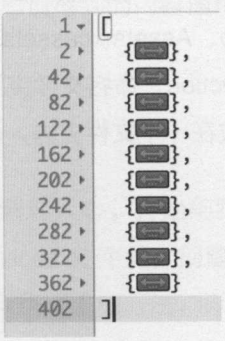


上图数据格式整齐多了。也许你并不清楚这些数据怎么阅读理解，没关系，接下来会加以解释。

- JSON -

JSON（JavaScript Object Notation），是一种轻量级的数据结构，处理速度快，易读又易写，并且常见于 API 返回的数据中。

Unsplash 的 API 返回数据也是 JSON 格式。回到 Postman 中，将所有数据收起，会看到返回的 JSON 数据格式是一个数组（Array），其中包含多组字典（Dictionary）。



在 Swift 中的表示应该是 [String: AnyObject]。

展开其中一个字典，会发现它定义了一张图片的各种信息。


```
{
  "id": "0l6qUieouA8",
  "created_at": "2016-08-16T01:18:39-04:00",
  "width": 6708,
  "height": 4472,
  "color": "#8B8972",
  "likes": 60,
  "liked_by_user": false,
  "user": {...},
  "current_user_collections": [],
  "urls": {...},
  "categories": [],
  "links": {...}
},
```


展开 urls，会看到这里又定义了同一张图片不同尺寸的 URL。


```
"urls": {
  "raw": "https://images.unsplash.com/photo-1471324652636-ea67fe86bce9",
  "full": "https://hd.unsplash.com/photo-1471324652636-ea67fe86bce9",
  "regular": "https://images.unsplash.com/photo-1471324652636-ea67fe86bce9?ixlib=rb-0.3.5&q=80&fm=jpg&crop=entropy&cs=tinysrgb&w=1080&fit
    =max&s=d789eecdd8266085d887e726c416bd7e",
  "small": "https://images.unsplash.com/photo-1471324652636-ea67fe86bce9?ixlib=rb-0.3.5&q=80&fm=jpg&crop=entropy&cs=tinysrgb&w=400&fit
    =max&s=19d43fea7c69a05c85f59bb80c01e1d9",
  "thumb": "https://images.unsplash.com/photo-1471324652636-ea67fe86bce9?ixlib=rb-0.3.5&q=80&fm=jpg&crop=entropy&cs=tinysrgb&w=200&fit
    =max&s=7aa8c2e50beccb12c24da4b4c93d7de1"
},
```


随意挑选一个 URL，在浏览器中打开会看到对应的图片。而我们需要做的就是把这张通过 API 获得的图片，在 Oslo 应用中正确显示出来。

- 储存 API 信息 -

首先把现在的文件结构梳理一下，以免后期乱七八糟。选中 Main.storyboard、ExifView.xib、StatisticsView.xib、Assets.xcassets 和 LaunchScreen.storyboard，右键单击 New Group from Selection，命名文件夹为 Assets。用同样的方法将 View Controller 和 cell 的 Swift 文件放在一个文件夹内，命名为 View Controllers。

右键单击黄色文件夹的 Oslo - New Group，命名文件夹为 Services。之后网络层的文件就都放在这个文件夹里。在这个文件夹上右键单击 NewFile，建立名为 Constants.swift 的文件，这里储存将要访问到的各种 API 的信息。

进入 Constants.swift，输入如下代码：

```
struct Constants {

  struct Base {
    static let UnsplashURL = "https://unsplash.com"
    static let UnsplashAPI = "https://api.unsplash.com"
    static let Curated = "/photos/curated"
    static let Authorize = "/oauth/authorize"
    static let Token = "/oauth/token"
  }

  struct Parameters {
    static let ClientID = ["client_id": "a1a50a27313d9bba143953469e415c24fc1096aea3be010bd46d4bd252a60896"]
    static let ClientSecret = ["client_secret": "c81b39a6a1f921a0b2b29de29f44fd176ffc101e816c5d2d34b6c951a885a68b"]
    static let RedirectURI = ["redirect_uri": "Oslo://photos"]
    static let GrantType = ["grant_type": "authorization_code"]
    static let ResponseType = ["response_type": "code"]
    static let Scope = ["scope": "public+read_user+write_likes"]
  }
}
```

在这里把需要用到的 API 分成两个部分，分离为 URL 和所携带的参数，这样的好处是在之后的逻辑部分更加明确。注意，这里的 ClientID 和 Secret 是笔者自己在 Unsplash 中申请的，请根据自己申请得到的内容信息进行替换。

- 建立网络层 -

现在开始建立网络层，有点激动人心。开始前我们要单独建立一个文件负责定义网络层的内容，右键单击 Services 文件夹，创建新文件 NerworkService.swift。在这个文件中主要定义一个发送网络请求的方法，以及如何处理请求错误的情况。

添加如下代码到文件内：

```
class NetworkService {
  enum HTTPMethod: String {
    case GET, POST, PUT, DELETE
  }
}
```

这里定义了发送请求时需要用到的方法，如果忘记了这些方法的用意，可以查看 HTTP Request Methods 一节。

接着在 class 内部建立如下方法：

```
class func request(url: URL,
                   method: HTTPMethod,
                   parameters: [[String: AnyObject]]? = nil,
                   headers: [String: String]? = nil,
                   completion: ((AnyObject) -> Void)? = nil) {

}
```

这便是发送网络请求的方法。其中需要传送的参数有要请求的 URL 地址、使用到的 HTTP 方法、URL 中需要携带的参数、以及获得数据后要做的反馈。

下面就来编写程序方法让它实现发送请求并获得数据。在方法内添加如下代码：

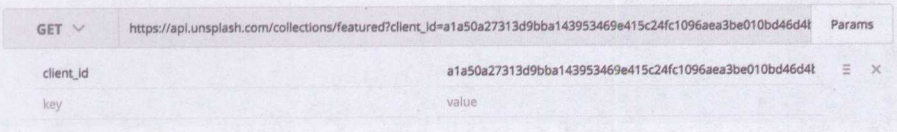
```
let session = URLSession.shared
```

首先建立了一个 session，session 概念在计算机领域广泛应用，可以理解为一个时间周期，也就是说告诉程序我需要一个时间的空档，才开始发送网络请求。

继续在下方添加代码：

```
let parsedURL = parse(url, with: parameters)
```

这段代码的意思是把需要传递的参数加在 URL 中，变成一个带有参数的 URL 地址。比如之前在 Postman 中，单击 URL 右方的 Params 按钮，参数是以字典的形式存在的：



这时 Xcode 会报错，因为我们还没有建立 parse 这个方法。在 class 内部最下方输入如下代码：

```
class func parse(_ url: URL, with parameters: [[String: AnyObject]]? = nil) -> URL {

  var components = URLComponents(url: url, resolvingAgainstBaseURL: false)!
  components.queryItems = [URLQueryItem]()

  if let params = parameters {
    for param in params {
      for (key, value) in param {
        let queryItem = URLQueryItem(name: key, value: "\(value)")
        components.queryItems!.append(queryItem)
      }
    }
  }

  return components.url!
}
```

解释这段代码，在 Playground 里更为方便。通过 File - New - Playground 新建一个 Playground（保存地址可以不在项目中），输入如下代码：

```
func parse(_ url: URL, with parameters: [[String: AnyObject]]? = nil) -> URL {

  var components = URLComponents(url: url, resolvingAgainstBaseURL: false)!
  components.queryItems = [URLQueryItem]()

  if let params = parameters {
    for param in params {
      for (key, value) in param {
        let queryItem = URLQueryItem(name: key, value: "\(value)")
        components.queryItems!.append(queryItem)
      }
    }
  }

  return components.url!
}

let url = URL(string: "https://api.unsplash.com/photos/curated")!
parse(url, with: [
  ["client_id": "a1a50a27313d9bba143953469e415c24fc1096aea3be010bd46d4bd252a60896" as AnyObject],
  ["page": 2 as AnyObject]
])
```

等待片刻后右侧会出现语句对应的结果：

一行行看下来，你很容易看出如何将参数加到给定的 URL 上。这也是 iOS API 很贴心的地方。

定义好这个方法后，Xcode 就不会报错了。有了 session，也有了要请求的 API，就缺少建立请求了，回到 request 方法最后部分，继续添加如下代码：

```
var request = URLRequest(url: parsedURL, cachePolicy: .reloadIgnoringLocalAndRemoteCacheData, timeoutInterval: 5.0)
request.httpMethod = method.rawValue
request.allHTTPHeaderFields = headers
```

以上定义了请求，并把参数中的 HTTP 方法和 header 附在请求中。

一切都准备好后，就可以定义一个请求任务（task），把 session 和 request 连接起来：

```
let task = session.dataTask(with: request) { (data, response, error) in

}
```

这里建立的任务是请求发送后会返回 data、response 和 error，供我们之后使用。不过我们还需要做一些容错机制：

```
guard error == nil else {
  print("An error occured: \(error)")
  return
}

guard let statusCode = (response as? HTTPURLResponse)?.statusCode, statusCode >= 200 && statusCode <= 299 else {
  print("Response code is not in range of 200 - 299.")
  return
}

guard let data = data else {
  print("No data returned.")
  return
}
```

这三种容错机制分别针对返回的 data、response 和 error。error 和 data 不难理解，重点说明 response。response 包含 HTTP 的 status code_，这个代码编号代表着发送请求后的不同状态，其中 200~299 这范围代表 response 是成功的。

在所有这些都保证没出错后，就可以使用服务端返回的数据了。不过这个数据并不是 JSON 格式的，要进行一次序列化。继续添加如下代码：

```
do {
  let result = try JSONSerialization.jsonObject(with: data, options: .allowFragments) as AnyObject
  completion(result)
} catch {
  print("Cannot parse data to JSON format.")
  return
}
```

如此我们通过 jsonObject(with data: Data, options opt: JSONSerialization.ReadingOptions = []) throws 这个方法将数据转化成了可用的 JSON 格式数据。

至此还没有结束，只是定义好这个请求任务，真正执行发送请求，还需要在 request 方法中最后加入如下代码：

```
task.resume()
```

这样网络层就建立好了。复习一下，网络层的编写顺序和结构如下：

- 建立 session。
- 对 URL 进行处理。
- 用处理好的 URL，建立 request。
- 用建立好的 session 和 request，建立 task。
- 处理各种可能的错误。
- 对数据进行 JSON 序列化。
- 启动 task。

- MVC -

在继续 MVC 之前，我们先来了解一下 iOS 开发的架构。

日常开发中有时会碰到一些小地方需要修改，结果要连带修改很多个关联文件，通常是由于最初编写代码时架构没有做好。架构是对程序代码的归类和组织。好的架构能够为程序的编写提供清晰的思路，还可以减轻后期维护的压力。

最广为人知的架构就是Apple官方使用的MVC(Model - View – Controller)架构。MVC 不仅适用于 iOS 开发，而且几乎在任何开发工作中都适用。在 MVC 的基础上，也有很多不同的演化，比如 MVVM、VIPER 等，本书着重介绍 MVC，其他架构及比较可以参考 https://swifting.io/blog/2016/09/07/architecture-wars-a-new-hope/ 这篇文章。

MVC 分离了数据（Model）和视图（View），通俗地说就是将看不见的数据部分按一定逻辑存放在一个地方，可见的视图部分存放在另外一个地方。至于控制器（Controller）则起着让数据和视图两者互相联系的纽带作用。比如在视图上单击红心，红心数据部分就需要被告知，然后加 1。“告知”就是控制器的任务。例如在 iOS 中，View Controller 扮演着控制器的角色，各种 Storyboard、xib 都可以看作是视图。

网络请求后会返回数据，而数据的更新则体现在视图上，这就需要考虑程序代码的架构情况。回到 Oslo 中，之前返回的 JSON 包含着各种各样的数据信息，因此需要建立一个文件夹来定义它们。

右键单击 Oslo 文件夹，选择 New File... - Swift File，将文件命名为 Photo.swift。添加如下代码：

```
import Foundation

struct Exif {
  let createTime: String
  let width: Int
  let height: Int
  let make: String
  let model: String
  let aperture: String
  let exposureTime: String
  let focalLength: String
  let iso: Int

  init(createTime: String,
       width: Int, height: Int,
       make: String,
       model: String,
       aperture: String,
       exposureTime: String,
       focalLength: String,
       iso: Int) {
    self.createTime = createTime
    self.width = width
    self.height = height
    self.make = make
    self.model = model
    self.aperture = aperture
    self.exposureTime = exposureTime
    self.focalLength = focalLength
    self.iso = iso
  }
}
```

我们需要把用到的数据都在代码里进行声明，并定义了初始化方法。这样返回的每一张图的数据都对应着每一个 Photo 数据。

进入 PhotosTableViewController.swift，在 class 内部第一行输入如下代码：

```
fileprivate var photos = [Photo]()
```

这样 controller 和 model 就连接起来了。同样，如下方法也需要进行相应的修改：

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
  return photos.count
}

override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
  let photo = photos[indexPath.row]
```

我们还要把网络返回的数据赋予 model。建立 OperationService.swift 文件，添加如下内容：

```
class OperationService {
  class func parseJsonWithPhotoData(_ data: [Dictionary<String, AnyObject>], completion: (_ photo: Photo) -> Void) {
    for jsonData in data {
      var id: String = ""
      var imageURL: String = ""
      var profileImageURL: String = ""
      var name: String = ""
      var userName: String = ""
      var isLike: Bool = false
      var heartCount: Int = 0
      var bio: String = ""
      var location: String = ""
      var portfolioURL: String = ""

      if let photoID = jsonData["id"] as? String {
        id = photoID
      }

      if let photoURLs = jsonData["urls"] as? [String: AnyObject],
        let photoURL = photoURLs["regular"] as? String {
        imageURL = photoURL
      }

      if let user = jsonData["user"] as? [String: AnyObject] {
        if let profileImage = user["profile_image"] as? [String: AnyObject],
          let mediumImage = profileImage["medium"] as? String {
          profileImageURL = mediumImage
        }

        if let fullName = user["name"] as? String {
          name = fullName
        }

        if let webName = user["username"] as? String {
          userName = webName
        }

        if let bioDescription = user["bio"] as? String {
          bio = bioDescription
        }

        if let locationDescription = user["location"] as? String {
          location = locationDescription
        }

        if let portfolioURLDescription = user["portfolio_url"] as? String {
          portfolioURL = portfolioURLDescription
        }
      }

      if let likeOrNot = jsonData["liked_by_user"] as? Bool {
        isLike = likeOrNot
      }

      if let likes = jsonData["likes"] as? Int {
        heartCount = likes
      }

      let photo = Photo(id: id,
                        imageURL: imageURL,
                        profileImageURL: profileImageURL,
                        name: name, userName: userName,
                        isLike: isLike,
                        heartCount: heartCount,
                        bio: bio,
                        location: location,
                        portfolioURL: portfolioURL)
      completion(photo)
    }
  }
}
```

看起来有些复杂，但实际上非常直接。首先网络请求返回的数据传到方法的参数 Data 中。注意，这里传入的数据是一个个的 Photo，不是 Photo 里面具体的各种数据。用 for jsonData in data 来访问每一个 Photo 数据。

各种 var 的部分用来声明我们需要的变量，各种 if 的部分则是从众多 Photo 信息中摘取我们需要的内容，赋给声明好的变量。

最后通过初始化把这些变量传到 Photo 的 model 中，完成 controller 接收数据，并给 model 传输数据的任务。不要忘了在 for...in... 的循环中，将 Photo 附到 photos 中。

有了 controller 给 model 传数据的逻辑，便可以发起网络请求，请求真实的数据并应用这套逻辑。将 viewDidLoad 的内容改为如下：

```
override func viewDidLoad() {
  super.viewDidLoad()

  tableView.rowHeight = UITableViewAutomaticDimension
  tableView.estimatedRowHeight = 400

  let url = URL(string: Constants.Base.UnsplashURL + Constants.Base.Curated)!

  NetworkService.request(url: url, method: NetworkService.HTTPMethod.GET,
                         parameters: [Constants.Parameters.ClientID as Dictionary<String, AnyObject>]) { jsonData in

    self.parseJson(with: jsonData)

    OperationQueue.main.addOperation {
      self.tableView.reloadData()
    }
  }
}
```

上图代码最先定义了要请求的 URL，然后将参数传入到 request 方法中，发起请求。如果仔细观察，会发现方法本身的 headers 参数和 completion 参数没有了。因 headers 在 Unsplash 的 API 文档中，获取精选照片的 API 并不需要传 header，所以这里就没有传。这里并不是没有 completion 了，而是在 Swift 中有 trailing closure 语法，completion 作为 closure 自然也可以这么用，jsonData in 中的 jsonData 就是 closure 需要传的参数，只是这样写看起来会更加简洁。

既然得到了返回的数据 jsonData，就可以将它通过 parseJson 传给刚刚写好的 Photo model。最后刷新 tableView 以此来应用最新的数据。

有了数据，下面就要把这些数据应用到界面上。首先在 NetworkService.swift 最后加入如下代码：

```
class func image(with imageURL: URL?, completion: @escaping (_ image: UIImage) -> Void) {
  DispatchQueue.global(qos: .userInitiated).async {
    if let imageURL = imageURL {
      do {
        let imageData = try Data(contentsOf: imageURL)
        guard let image = UIImage(data: imageData) else { return }

        DispatchQueue.main.async {
          completion(image)
        }
      } catch {
        print("Image download failed.")
      }
    }
  }
}
```

这个方法定义了异步获取网络图片的逻辑，这里不太明白没关系，下一个章节后再看这个方法就能明白很多。

回到 PhotosTableViewController.swift，界面是由每一个 cell 组成的，所以定位到 tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell 这个方法中。先显示照片部分，在 cell.delegate = self 下方输入如下代码：

```
if let photoURL = URL(string: photo.imageURL) {
  NetworkService.image(with: photoURL) { image in
    cell.photoImageView.image = image
  }
}
```

首先用之前储存到 Photo Model 中的数据，将图片的 URL 取出，并转化成 URL 类型。有了 URL，就可以发送请求来获得实际的图片数据了（图片数据后缀通常为 .png、.jpg、.gif、.webp）。这里用的是 image（with imageURL: URL?, completion: @escaping (_ image: UIImage) -> Void）这个方法，它作为一个 class 方法写在 NetworkService class 中。关于这个方法的说明我们在下一个章节讲解，现在先简单地调用它，知道它的作用是根据 URL 发送请求，拿到图片数据，并在 closure 中自定义执行即可。在这里，我们直接把取到的图片赋给 cell 图片的值。

同样的方法用于用户头像。在上面代码下方继续添加如下代码：

```
if let photoImageURL = photo.imageURL, let photoURL = URL(string: photoImageURL) {
  NetworkService.image(with: photoURL) { image in
    cell.photoImageView.image = image
  }
}
```

最后将几个 label 赋值：

```
cell.userLabel.text = photo.name
cell.isLike = photo.isLike
cell.heartCountLabel.text = "\(photo.heartCount)"
```

Build 一下。呈现出活灵活现的数据！不得不说，这是你在 iOS 开发史上迈出的一大步。

- Grand Central Dispatch & OperationQueue -

之前的代码中有几处没有讲解，分别为 OperationQueue 和 DispatchQueue。这两个看似和业务代码逻辑并无瓜葛的语句是什么呢？下面一点一点来说明。

在不断提升效率的时代，计算机也进入了多核（Multi-core）多线程（Multi-thread）时代。也许你并不清楚线程是什么，简单说来，线程就是所处理的一项任务，比如你在看电视就是一个线程。多线程的并发（Concurrent）执行，就好似你一边看电视一边吃薯片。而如何让看电视和吃薯片更好地同时进行（你不希望低头找大块薯片的时候错过了关键的情节），这就是线程控制的工作了。

在程序应用中也一样，在玩手机刷内容的同时，你肯定不想因为网络原因图片加载不出来导致整个应用卡壳，影响你做其他事情。因此线程的控制就变得至关重要。

在 iOS 开发中，线程的控制是通过 GCD（Grand Central Dispatch）和 OperationQueue 来进行的。这里首先介绍 GCD。在 Swift 3 以前，GCD 的 API 是看起来古老而神秘，比如 dispatch_once_t 这种看上去让人有点望而却步的名字。当多了解 GCD 的来源后就觉得这种感觉无可厚非，原来 GCD 源自于 C 语言，由低级的 API 转化而来，用来控制线程执行的优先级、归属等。到了 Swift 3，GCD 这部分 API 变得更加 Swift 了，写起来、读起来都顺手顺口了很多。

我们并不希望图片传回所发生的各种未知情况干扰到主线程，因为那样会造成界面卡死的悲剧，所以这部分工作使用异步线程，让图片的传输和显示什么时候准备好什么时候执行，不要影响到应用的其他逻辑。回到 NetworkService.swift 的 image(with imageURL: URL?, completion: @escaping (_ image: UIImage) ->

Void) 方法中，第一行 DispatchQueue.global(qos: .userInitiated).async 的意思就是在全局线程中打开一个异步线程，其中 qos 为优先级，具体的实例可以参照 http://www.appcoda.com/grand-central-dispatch/ 这篇文章。拿到图片文件后，要传给 completion 的 closure 中，这里用 DispatchQueue.main.async 在主线程中又开了一个异步的线程，意思是图片的处理也是异步线程的。这样在快速滑动 table view 时，图片的接收和显示就不会卡在界面。因为图片是比较大的数据，因此在进行图片相关的处理时，用异步线程来处理是很合适的。

GCD 是操作线程的，那 OperationQueue 又是什么呢？也是操作线程的。其实 OperationQueue 内部还是 GCD，只是它是在 GCD 上面一层更高级的 API，还对应了比如 start()、cancel() 这样方便的方法。至于什么时候用 GCD，什么时候用 OperationQueue，见仁见智。

再回到 PhotosTableViewController.swift 的 OperationQueue.main.addOperation 中，在 JSON 数据返回时，主线程将 Table View 刷新，以应用最新的数据。

尽管我们可以对线程进行干涉，但在实际中，建议少进行类似的操作。因为线程的操作一旦出现问题很难查找，会难以复现，可怕的是这类的操作还经常容易出现问题，即使是一个老练的程序员，恐怕也无法保证线程的安全。

- 缓存 -

在模拟器中上下划动，会发现本来已经加载好的图片又重新加载一遍，这种体验实在不怎么样。对于已经获得的数据，怎么才能保存呢？缓存（Cache）是一个办法。其实有很多方法能够存储已经得到的数据，比如 Core Data 等。这里之所以使用缓存是因为我们得到的数据不必“永久”保存，而缓存正是为这样的数据所设计的。在 iOS 中，NSCache 也无法保证数据的可靠，可能在系统其他地方需要内存时，这部分数据就被清除了。

在 PhotosTableViewController class 内部继续声明如下变量来保存缓存数据：

```
fileprivate var photoCache = NSCache<NSString, UIImage>()
```

缓存的存储有点类似于字典，都是 key - value 存储的。这里将 key 的类型定义

为 NSString，value 的类型定义为 UIImage。定义好缓存就可以将下载好的数据存储进去，在 tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell 中将下面代码：

```
if let photoImageURL = photo.imageURL, let photoURL = URL(string: photoImageURL) {
  NetworkService.image(with: photoURL) { image in
    cell.photoImageView.image = image
  }
}
```

替换为：

```
if let photoImageURL = photo.imageURL, let photoURL = URL(string: photoImageURL) {
  if let cachedImage = self.photoCache.object(forKey: photoImageURL as NSString) {
    cell.photoImageView.image = cachedImage
  } else {
    NetworkService.image(with: photoURL) { image in
      self.photoCache.setObject(image, forKey: photoImageURL as NSString)

      cell.photoImageView.image = image
    }
  }
}
```

这里真正加入的只是一个 if...else... 的条件语句。首先判断是否有 key 为 photoImageURL 的缓存数据，如果有，就将对应的值赋给 cell 的图片；如果没有，就把这个 key 和对应的图片数据加到缓存中，并让 cell 显示该图片。

```
if let profileImageURL = photo.imageURL, let profileURL = URL(string: profileImageURL) {
  NetworkService.image(with: profileURL) { image in
    cell.userImageView.image = image
  }
}
```

替换为：

```
if let profileImageURL = photo.profileImageURL, let profileURL = URL(string: profileImageURL) {
  if let cachedImage = self.photoCache.object(forKey: profileImageURL as NSString) {
    cell.userImageView.image = cachedImage
  } else {
    NetworkService.image(with: profileURL) { image in
      self.photoCache.setObject(image, forKey: profileImageURL as NSString)

      cell.userImageView.image = image
    }
  }
}
```

没问题后就 Build 一下吧，这次已经获取到的图片就不会重新去请求并显示了。

有一个小问题，每次划动到新的 cell 时都会显示一下其他的图片。这个需要在 if letphotoImageURL = photo.imageURL, let photoURL = URL(string: photoImageURL) 前加入如下代码，让 cell 显示前图片先置空：

```
cell.photoImageView.image = nil
cell.userImageView.image = nil
```

然后将 cell.photoImageView.image = image 一行替换为：

```
if let updateCell = tableView.cellForRow(at: indexPath) as? PhotoTableViewCell {
  updateCell.photoImageView.image = image
}
```

再 Build 一下，就没问题了。

- 下拉刷新 & 划动加载 -

每次打开应用只能看到 10 张图片未免太无聊了，要想制作出让人尽情的应用，丰富的内容必不可少，下面是实现两个最常见的加载数据方式。

首先是下拉刷新。在 iOS 10 以前实现下拉刷新的方式多种多样，基本是以下拉的视图位置来判断刷新时机。好消息是从 iOS 10 开始下拉刷新被写在了 iOS 官方的 API 中，它被定义在了 UIScrollView 中：

```
@available(iOS 10.0, *)
open var refreshControl: UIRefreshControl?
```

当然，UITableView 继承了 UIScrollView，因此也可以无压力使用这个 API。实现起来也非常简单。首先在顶部声明 photoCache 的下方添加如下代码：

```
lazy var feedRefreshControl: UIRefreshControl = {
  let feedRefreshControl = UIRefreshControl()
  feedRefreshControl.addTarget(self, action: #selector(load(with:)), for: .valueChanged)

  return feedRefreshControl
}()
```

feedRefreshControl 定义了刷新的操作，之后可以在任何部分使用它，开始刷新、停止刷新等。feedRefreshControl.addTarget(self, action: #selector(load(with:)), for: .valueChanged) 意思是当刷新发生时，什么方法会被调用，这里定义的方法

是 load(with:)，会发生报错是因为我们还没有开始定义它。currentPage 是为了不断加载内容定义的页码，从 https://unsplash.com/documentation#list-photos 的 Parameters 中可以看到，Unsplash 提供的数据加载是分页式的，每页默认只返回 10 条数据。

将 viewDidLoad 中代码内容替换为如下：

```
override func viewDidLoad() {
  super.viewDidLoad()

  tableView.rowHeight = UITableViewAutomaticDimension
  tableView.estimatedRowHeight = 400

  currentPage = 1

  if #available(iOS 10.0, *) {
    tableView.refreshControl = feedRefreshControl
  } else {
    tableView.addSubview(feedRefreshControl)
  }

  load()
}
```

首先在每次进入 tableview 时，分页的页数都回到第一页。接着把 feedRefreshControl 和 tableview 联系起来。由于 open var refreshControl: UIRefreshControl? 是 iOS 10 的接口，所以这里要加上条件的判断，如果不是 iOS10 的系统，则要把 feedRefreshControl 加在 tableview 视图上。

接下来就定义关键的 load(with:)，在 tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell 下方加入如下代码：

```
func load(with page: Int = 1) {
  feedRefreshControl.beginRefreshing()

  let url = URL(string: Constants.Base.UnsplashAPI + Constants.Base.Curated)!

  NetworkService.request(url: url, method: NetworkService.HTTPMethod.GET,
                         parameters: [
                          Constants.Parameters.ClientID as Dictionary<String, AnyObject>,
                          ["page": page as AnyObject]
  ]) { jsonData in
    OperationService.parseJsonWithPhotoData(jsonData as! [Dictionary<String, AnyObject>]) { photo in
      self.photos.append(photo)
      self.personalPhotos.append(nil)
      self.profileImages.append(nil)
    }

    OperationQueue.main.addOperation {
      self.tableView.reloadData()
      self.feedRefreshControl.endRefreshing()
    }
  }
}
```

with page: Int = 1 参数的意思是让默认加载第一页的数据，这样在每次使用

viewDidLoad 方法时，直接用 load() 就可以了，只有在不断下划加载数据的时候，才需要改变这个数字。

> 运行错误（Runtime error）和编译错误（Compile error）
>
> 运行错误是程序在运行时报的错。如程序是正常运行，但是会出现闪退等问题。某些错误在编写程序的时候 IDE（Integrated Development Environment）是不会及时报错的，比如某个字符串拼错、访问的下标超过了容量等。注意，这种错误在 Playground 中会报错，因为 Playground 是编译后立即运行的。
>
> 编译错误就比较常见了。比如写着写着程序突然出现了红色的感叹号，提示 API 使用错误、语法错误等。这类错误出现后是无法 Build 应用的，必须修复后才能运行。

之所以这里给数据数组填充 nil 值，是因为让这两个数组拥有足够的容量（Capacity），这样之后才可以访问下标正常赋值，否则会报 Array is out of index 的运行错误。

与之前 request 方法不同的是，这里的 parameters 中多传了一个 ["page": page as AnyObject]，这样在发送请求的时候就可以带上 page 参数，请求不同的页码。

最后将数据赋给 photos，重新加载 tableview，以及停止刷新。

Build 一下，下拉刷新没问题啦！

下拉刷新完成后，就可以开始数据划动加载，幸好这一点也不复杂。在 tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell 下方加入如下代码即可：

```
override func tableView(_ tableView: UITableView, willDisplay cell: UITableViewCell, forRowAt indexPath: IndexPath) {
  if indexPath.row == photos.count - 1 {
    currentPage += 1

    load(with: currentPage)
  }
}
```

接口本身的意思是用来定义 cell 将要显示出来的逻辑，听上去很像下划预加载数据的感觉。并且这里的代码也很好理解——当 tableview 的 cell 行数和当前数据的

数量一样时，就请求新的一页数据继续加载。

再 Build 一下，是不是随着划动在不断加载数据。

- 用 Segue 传输数据 -

如果注意观察会发现，Profile View Controller 中的用户头像、用户名、用户描述等信息，与 Personal Photo View Controller 中的图片和 Photos View Controller 共用的是一套数据。因此在进入 Profile View Controller 或 Personal Photo View Controller 时不必重新发送网络请求去请求同样的一份数据。如果前一个 View Controller 中有已经下载好的数据，可以直接传输给下一个 View Controller，不过怎么传输呢?

想到 Segue 连接了各个 View Controller，那么 Segue 是不是也可以进行 View Controller 之间数据的传输呢? 答案是肯定的。下面我们就用 Segue 把 Photos View Controller 获得的数据传输给 Profile View Controller。

首先在 ProfileViewController.swift 中，class 内部顶部声明如下两个变量，用来接受 photo model 和下载好的图片：

```
var photo: Photo!
var profileImage: UIImage!
```

然后进入 PhotosViewController.swift，在 Photos 变量下加入如下代码用来存储下载好的图片：

```
fileprivate var profileImages = [UIImage?]()
```

找 到 func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell 中的 cell.userImageView.image = image 一行，在上面加入：

```
self.profileImages[indexPath.row] = image
```

将下载好的 image 数据赋给 profileImage 数组。注意，这里被赋值的数组下标应该是该 cell 所在的行数。

在传输数据之前，先定义一个方法来定位到触发元素所在的 cell，并读取到 cell 的行数，这样才可以访问到正确的数据。在 tapToPerformSegue(_ sender: AnyObject) 方法下方加入如下代码：

```
func getCurrentCellRow(sender: Any?) -> Int? {
  if let sourceSender = sender as? UITapGestureRecognizer,
    let cellView = sourceSender.view?.superview,
    let cell = cellView.superview as? PhotoTableViewCell {
    let selectedIndexPath = tableView.indexPath(for: cell)!.row

    return selectedIndexPath
  } else {
    return nil
  }
}
```

iOS API 中的 func prepare(for segue: UIStoryboardSegue, sender: Any?) 可实现我们用 segue 来传输数据的目的。在 func parseJson(with data: [AnyObject]) 上方加入如下代码：

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  guard let selectedIndexPathRow = getCurrentCellRow(sender: sender) else { return }

  if segue.identifier == "ProfileSegue" {
    if let destinationViewController = segue.destination as? ProfileViewController {
      destinationViewController.photo = photos[selectedIndexPathRow]
      destinationViewController.profileImage = profileImages[selectedIndexPathRow]
    }
  }
}
```

如果触发的 segue 为 ProfileSegue，且 segue 要过渡到的 View Controller 是 ProfileViewController，那么把 PhotosViewController 中该 cell 对应的 photo model 传给 ProfileViewController 中的 photo model，并将对应的 profileImage 数据传给 ProfileViewController 中的 profileImage。

这时 ProfileViewController 中的 Photo 和 profileImage 都拥有了数据，就可以更新 UI。在最后一个 @IBOutlet 下方加入如下代码：

```
override func viewDidLoad() {
  super.viewDidLoad()

  avatarImageView.image = profileImage
  userLabel.text = photo.name
  bioLabel.text = photo.bio

  if photo.location != "" {
    locationLabel.text = photo.location
  } else {
    locationLabel.text = "No Location"
  }

  if photo.portfolioURL.contains("instagram.com") {
    portfolioImage.image = instagram
    let instagramName = photo.portfolioURL.components(separatedBy: "/")
    portfolioName.setTitle(instagramName[3], for: .normal)
  } else {
    portfolioImage.image = portfolio
    portfolioName.setTitle("Website", for: .normal)
  }
}
```

将数据分别赋给 UI 上的每一个组件。在 Unsplash API 返回的数据中，portfolio_url 有的是 Instagram 的链接和其他网站的链接，这里我们区分对待一下——如果是 Instagram 的链接，也就是链接中包含“instagram.com”，那么将用户名的部分赋给 UI。如果是其他的链接，UI 则统一显示“Website”。

Build 一下，是不是从 Photos View Controller 到 Profile View Controller 时并没有发送特别的网络请求，加载图片也没有浪费时间?

ProfileViewController 中 CollectionView 的数据和 PhotosViewController 显示图片的方法大同小异。在 var profileImage: UIImage! 下方加入这三个变量，分别存放用户的照片 URL、照片缓存和下载照片：

```
fileprivate var photoURLs = [String]()
fileprivate var photoCache = NSCache<NSString, UIImage>()
fileprivate var personalPhotos = [UIImage?]()
```

如何获取用户的图片呢? 查看 https://unsplash.com/documentation#list-a-users-photos 发现 API 为 GET /users/:username/photos，其中 :username 根据用户的不同而不同。

首先定义一个变量保存用户发布的图片总数：

```
fileprivate var totalPhotosCount: Int = 0
```

在 class 内部定义 load 方法：

```
func load(with page: Int = 1) {
  let urlString = Constants.Base.UnsplashAPI + "/users/\(photo.userName)/photos"
  let url = URL(string: urlString)!

  NetworkService.request(url: url,
                         method: NetworkService.HTTPMethod.GET,
                         parameters: [Constants.Parameters.ClientID as Dictionary<String, AnyObject>]) { jsonData in
                          guard let data = (jsonData as? [Dictionary<String, AnyObject>]) else { return }

                          let firstData = data[0]

                          if let user = firstData["user"] as? [String: AnyObject],
                            let totalPhotos = user["total_photos"] as? Int {
                            self.totalPhotosCount = totalPhotos
                          }

                          OperationService.parseJsonWithPhotoData(jsonData as! [Dictionary<String, AnyObject>]) { photo in
                            self.personalPhotos.append(photo)
                            self.downloadedPersonalPhotos.append(nil)
                          }

                          OperationQueue.main.addOperation {
                            self.collectionView.reloadData()
                          }
  }
}
```

在 func viewDidLoad() 最后加入：

```
load()
```

为了实现横向划动不断加载，在 fileprivate var personalPhotos = [UIImage?]() 下方加入：

```
fileprivate var currentPage = 1
```

在 func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell 下方加入：

```
func collectionView(_ collectionView: UICollectionView, willDisplay cell: UICollectionViewCell, forItemAt indexPath: IndexPath) {
  if indexPath.row == personalPhotos.count - 1 && indexPath.row != totalPhotosCount - 1 {
    currentPage += 1

    load(with: currentPage)
  }
}
```

Build 一下，就能显示用户的照片了！

既然已经读取图片和红心数据了，顺便把 Personal Photo View Controller 中的图片和红心数据解决了吧。在 PersonalPhotoViewController class 内部加入：

```
var photo: Photo!
var personalPhoto: UIImage?
```

这里之所以把 personalPhoto 声明为 Optional 是因为图片内存都比较大，根据网络环境的不同，可能会出现前一个 View Controller 没有下载完数据就进入到另一个 View Controller 中，因为没有数据 segue 也就不会传递数据。

为 PhotosViewController.swift 的 func prepare(for segue: UIStoryboardSegue, sender: Any?) 内部最外层 if 加入如下 else if 代码：

```
else if segue.identifier == "PhotoSegue" {
  if let destinationViewController = segue.destination as? PersonalPhotoViewController {
    destinationViewController.photo = photos[selectedIndexPathRow]
    destinationViewController.personalPhoto = personalPhotos[selectedIndexPathRow]
  }
}
```

这时 PersonalPhotoViewController 便有了数据，可以更新 UI。在最后一个 @IBOutlet 下方加入：

```
override func viewDidLoad() {
  super.viewDidLoad()

  if let downloadedImage = personalPhoto {
    personalPhotoImageView.image = downloadedImage
  } else {
    personalPhotoImageView.image = nil

    let imageURL = URL(string: photo.imageURL)
    NetworkService.image(with: imageURL) { image in
      self.personalPhoto = image
      self.personalPhotoImageView.image = image
    }
  }

  photo.isLike ? heartButton.setBackgroundImage(♥heart-liked, for: .normal) : heartButton.setBackgroundImage(♡heart-outline, for: .normal)

  heartCountLabel.text = String(photo.heartCount)

  savePhotoLabel.alpha = 0
}
```

Build 一下，单击图片进入图片页面，显示正常。注意，这里还处理了刚才讲的网络原因可能导致没有图片数据传过来的情况。在这种情况下，需要通过 NetworkService.image(with:) 这个方法再次发送网络请求得到图片数据。

这样从 PhotosViewController 的数据接收就全部完成了。切换到 ProfileViewController.swift 中，在 func collectionView(_ collectionView: UICollectionView, didSelectItemAt indexPath: IndexPath) 下方加入：

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "PhotoSegue" {
    if let selectedIndexPath = collectionView.indexPathsForSelectedItems?[0].row,
      let destinationViewController = segue.destination as? PersonalPhotoViewController {
      destinationViewController.photo = personalPhotos[selectedIndexPath]
      destinationViewController.personalPhoto = downloadedPersonalPhotos[selectedIndexPath]
    }
  } else if segue.identifier == "PortfolioSegue" {
    if let destinationViewController = segue.destination as? PortfolioWebViewController {
      destinationViewController.navigationItem.title = "\(photo.name)'s website"
      destinationViewController.portfolioURL = photo.portfolioURL
    }
  }
}
```

destinationViewController.portfolioURL = photo.portfolioURL 一行是将 portfolio URL 传给 Web View 的 View Controller，才可以加载网页。

建立 PortfolioWebViewController.swift，关联 Storyboard 中的 View Controller，并连接组件到代码：

```
@IBOutlet var portfolioWebView: UIWebView!
```

在下方继续添加变量来接收 URL：

```
var portfolioURL = ""
```

至于怎么加载网页，在之后的章节会介绍。

大功告成！写了不少东西，赶快 Build 一下吧！

- 更新 xib 信息 -

还记得 ExifView.xib 和 StatisticsView.xib 这两个文件吗？从 API 上读取数据后怎样才能更新上面的信息呢？其实和 storyboard 的 UI 更新是一个意思。

观察一下 https://unsplash.com/documentation#get-a-photo 和 https://unsplash.com/documentation#get-a-photos-stats 能知道访问这些数据所需要的两个 API 为 /photos/:id 和 /photos/:id/stats。建立两个 model 来储存数据。在 Models 文件夹 -Photo 上单击右键 New File…建立 Exif.swift 和 Statistics.swift 两个文件。内容分别如下：

Exif. swift

```
import Foundation

struct Exif {
  let createTime: String
  let width: Int
  let height: Int
  let make: String
  let model: String
  let aperture: String
  let exposureTime: String
  let focalLength: String
  let iso: Int

  init(createTime: String,
       width: Int, height: Int,
       make: String,
       model: String,
       aperture: String,
       exposureTime: String,
       focalLength: String,
       iso: Int) {
    self.createTime = createTime
    self.width = width
    self.height = height
    self.make = make
    self.model = model
    self.aperture = aperture
    self.exposureTime = exposureTime
    self.focalLength = focalLength
    self.iso = iso
  }
}
```

Statistics. swift

```
import Foundation

struct Statistics {
  let downloads: Int
  let views: Int
  let likes: Int

  init(downloads: Int, views: Int, likes: Int) {
    self.downloads = downloads
    self.views = views
    self.likes = likes
  }
}
```

这两个文件分别声明了各自需要的变量，并提供了初始化方法。

要访问 API 的数据，就涉及对 JSON 数据处理的问题。进入 OperationService. swift，在 class 内部最下方加入下面两个方法，分别处理 Exif 数据和 Statistics 数据，

和之前处理 Photo 数据差不多：

```
class func parseJsonWithExifData(_ data: Dictionary<String, AnyObject>, completion: (_ exifData: Exif) -> Void) {
    var createTime: String = ""
    var width: Int = 0
    var height: Int = 0
    var make: String = ""
    var model: String = ""
    var aperture: String = ""
    var exposureTime: String = ""
    var focalLength: String = ""
    var iso: Int = 0

    if let photoCreatedTime = data["created_at"] as? String {
      createTime = photoCreatedTime
    }

    if let photoWidth = data["width"] as? Int {
      width = photoWidth
    }

    if let photoHeight = data["height"] as? Int {
      height = photoHeight
    }

    if let photoExif = data["exif"] as? [String: AnyObject] {
      if let photoMake = photoExif["make"] as? String {
        make = photoMake
      }

      if let photoModel = photoExif["model"] as? String {
        model = photoModel
      }

      if let photoAperture = photoExif["aperture"] as? String {
        aperture = photoAperture
      }

      if let photoExposureTime = photoExif["exposure_time"] as? String {
        exposureTime = photoExposureTime
      }

      if let photoFocalLength = photoExif["focal_length"] as? String {
        focalLength = photoFocalLength
      }

      if let photoISO = photoExif["iso"] as? Int {
        iso = photoISO
      }
    }

    let exifInfo = Exif(createTime: createTime,
                        width: width,
                        height: height,
                        make: make,
                        model: model,
                        aperture: aperture,
                        exposureTime: exposureTime,
                        focalLength: focalLength,
                        iso: iso)
    completion(exifInfo)
}

class func parseJsonWithStatisticsData(_ data: Dictionary<String, AnyObject>, completion: (_ statisticsData: Statistics) -> Void) {
    var downloads: Int = 0
    var views: Int = 0
    var likes: Int = 0

    if let downloadsNumber = data["downloads"] as? Int {
      downloads = downloadsNumber
    }

    if let viewsNumber = data["views"] as? Int {
      views = viewsNumber
    }

    if let likesNumber = data["likes"] as? Int {
      likes = likesNumber
    }

    let statisticsInfo = Statistics(downloads: downloads, views: views, likes: likes)
    completion(statisticsInfo)
}
```

返回到 PersonalPhotoViewController.swift，在 isLike 变量下方加入：

```
fileprivate var exifInfo: Exif?
fileprivate var statisticsInfo: Statistics?
```

使用上面两个变量来接收 API 返回的数据。其实这里可以将这两个变量的作用域（Scope）设置成 private，但因为这是 API 返回的数据，希望在以后的改版中其他地方会用到，所以还是设置成 fileprivate 吧。

有了接收 API 返回的数据 model，现在就缺一个发送请求的方法了，在 class 内

部最后加入：

```
private func load() {

    let statisticsURL = URL(string: Constants.Base.UnsplashURL + "/photos/" + photo.id + "/stats")!

    NetworkService.request(url: statisticsURL, method: NetworkService.HTTPMethod.GET,
                           parameters: [Constants.Parameters.ClientID as Dictionary<String, AnyObject>]) { jsonData in
                            OperationService.parseJsonWithStatisticsData(jsonData as! Dictionary<String, AnyObject>) {
                                statisticsInfo in
                                self.statisticsInfo = statisticsInfo
                            }
    }

    let exifURL = URL(string: Constants.Base.UnsplashURL + "/photos/" + photo.id)!

    NetworkService.request(url: exifURL, method: NetworkService.HTTPMethod.GET,
                           parameters: [Constants.Parameters.ClientID as Dictionary<String, AnyObject>]) { jsonData in
                            OperationService.parseJsonWithExifData(jsonData as! Dictionary<String, AnyObject>) { exifInfo in
                                self.exifInfo = exifInfo
                            }

    }
}
```

首先这里定义了要发送的 API 地址，然后用 request 方法发送请求，将返回的数据进行解析，最后将 model 赋给 class 内部的数据 model，一切都按流程办事。

数据代码完成后就要调用方法。在 func viewDidLoad() 最下方加入：

```
load()
```

当然也可以选择在单击显示 ExifView.xib 或 StatisticsView.xib 时调用，可利用用户看照片的时间，为用户加载好后面的数据，用户需要时瞬间就会呈现，因数据内存不大需要的流量很少。

在 personalPhotoImageView.addSubview(statisticsView!) 上方加入：

```
let numberFormatter = NumberFormatter()
numberFormatter.numberStyle = .decimal

statisticsView!.downloadsLabel.text = numberFormatter.string(from: statisticsInfo!.downloads as NSNumber)
statisticsView!.viewsLabel.text = numberFormatter.string(from: statisticsInfo!.views as NSNumber)
statisticsView!.likesLabel.text = numberFormatter.string(from: statisticsInfo!.likes as NSNumber)
```

在 personalPhotoImageView.addSubview(exifView!) 上方加入：

```
let dateTime = exifInfo!.createTime.components(separatedBy: "T")[0]
let dateFormatter = DateFormatter()
dateFormatter.dateFormat = "yyyy-MM-dd"
let date = dateFormatter.date(from: dateTime)
dateFormatter.dateStyle = .long
let createDate = dateFormatter.string(from: date!)
exifView!.createdTimeLabel.text = createDate

exifView!.dimensionsLabel.text = "\(exifInfo!.width) x \(exifInfo!.height)"
exifView!.makeLabel.text = exifInfo!.make
exifView!.modelLabel.text = exifInfo!.model
exifView!.apertureLabel.text = exifInfo!.aperture
exifView!.exposureTimeLabel.text = exifInfo!.exposureTime
exifView!.focalLengthLabel.text = exifInfo!.focalLength
exifView!.isoLabel.text = "\(exifInfo!.iso)"
```

这样就更新了 UI 的信息。这里 numberFormatter 的作用是让数字在千分位上加个逗号，如 1234 显示为 1,234，这和 Unsplash 官网所显示的方式统一；dateFormatter 的作用是让返回的时间字符串经过处理后显示为智能可读的形式，比如 2016-11-30 显示为“Nov,30,2016”。

完成后 build 一下吧，这下就能看到图片的参数和更多的信息了！

- OAuth 2 与登录 -

只有用户登录后，才能知道“喜欢”的操作对应哪个用户，从而更新那部分数据。用户如何在应用内实现登录，在编写登录代码之前，要先清楚 OAuth2 的概念，也就是客户端和服务端背后的认证逻辑。在 https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2 这篇文章中有非常详尽的解释，这里只简单介绍一下。OAuth 2 的用户认证步骤应遵循以下顺序：

1. 客户端先发起请求告诉服务端，要做用户认证这件事。这一步的请求链接由以下几个部分组成，如下图所示。

endpoint	请求的基础地址
client_id	应用的ID，在申请 API 访问时会自动生成
redirect_uri	当服务端返回授权码后，客户端要跳转到的地方
response_type	告诉服务端你想要的响应类型
scope	需要用户授权哪些权限

2. 请求发送后，服务端会要求用户授权。如果用户没有登录，会要求用户先登录，然后再自动跳转到授权页面。注意这里的登录是在 Unsplash 官网的登录，而不是在应用里。

3. 用户授权后，服务端会返回授权码（authorization code），说明这里的应用是服务端认识的，是安全的，所以给了你授权码。

4. 有了授权码，就可以用授权码发送新的请求去获取 token，每一个用户对应一个 token，可以看作是“用户码”。获得 token 的请求由以下几个部分组成，如下图所示。

endpoint	请求的基础地址
client_id	应用的 ID，在申请 API 访问时会自动生成
client_secret	应用的密钥，在申请 API 访问时会自动生成
redirect_uri	当服务端返回授权码后，客户端要跳转到的地方
code	服务端返回过来的授权码
grant_type	服务端的授权类型

这个请求发送后，服务端就会返回 token。有了 token，相当于用户已经登录。之后和用户信息相关的操作，比如添加喜欢的应用、更新用户头像等的请求参数中带着 token 就可以了。

明白了 OAuth 2 流程后，回顾一下 Unsplash 是怎样应用这个流程的。进入 https://unsplash.com/documentation#user-authentication，这里有详尽的说明如何使用 OAuth 2 进行 Unsplash 的用户认证。如果忘记了申请时应用的信息，可以在 https://unsplash.com/oauth/applications（为了方便，可命名这个页面为应用信息页面）中单击应用名称进入后查看。结合刚才的流程，大致如下图所示。

获取 授权码

endpoint	https://unsplash.com/oauth/authorize
client_id	应用信息页面中的 Application ID
redirect_uri	应用信息页面中的 Callback URLs 这里之前添加的是本地测试的地址，我们也可以添加一个自定义的地址，这里我添加的是 Oslo://photos。需要注意的是如果自定义，格式应该为 应用名称://页面 。
response_type	Unsplash 要求这里的值为 code
scope	public+read_user+write_likes，对应为用户的公开信息、用户的私人信息和添加/取消用户喜欢的照片。更多的权限可以在 Permission scopes 里查看。

获取 token

endpoint	https://unsplash.com/oauth/token
client_id	应用信息页面中的 Application ID
client_secret	应用信息页面中的 Secret
redirect_uri	Oslo://photos 或者是你自己定义的
code	获取授权码的请求返回的授权码
grant_type	Unsplash 要求这里的值为 authorization_code

消化一下这些概念，然后就可以实际操作了。

首先重新复习一遍 Constants.swift 中的常量：

```
struct Constants {

  struct Base {
    static let UnsplashURL = "https://unsplash.com"
    static let UnsplashAPI = "https://api.unsplash.com"
    static let Curated = "/photos/curated"
    static let Authorize = "/oauth/authorize"
    static let Token = "/oauth/token"
  }

  struct Parameters {
    static let ClientID = ["client_id": "a1a50a27313d9bba143953469e415c24fc1096aea3be010bd46d4bd252a60896"]
    static let ClientSecret = ["client_secret": "c81b39a6a1f921a0b2b29de29f44fd176ffc101e816c5d2d34b6c951a885a68b"]
    static let RedirectURI = ["redirect_uri": "Oslo://photos"]
    static let GrantType = ["grant_type": "authorization_code"]
    static let ResponseType = ["response_type": "code"]
    static let Scope = ["scope": "public+read_user+write_likes"]
  }
}
```

现在看是不是很熟悉了？记住，请将 ClientID 和 ClientSecret 更换为自己的。

有了这些，就可以发送获取授权码的请求了。用户希望这个页面一出现就开始请求，进入 LoginViewController.swift，在 func viewDidLoad() 最下方添加：

```
let authorizationURL = NetworkService.parse(URL(string: Constants.Base.UnsplashURL + Constants.Base.Authorize)!, with: [
    Constants.Parameters.ClientID as Dictionary<String, AnyObject>,
    Constants.Parameters.ResponseType as Dictionary<String, AnyObject>,
    Constants.Parameters.RedirectURI as Dictionary<String, AnyObject>,
    Constants.Parameters.Scope as Dictionary<String, AnyObject>
  ])

let request = URLRequest(url: authorizationURL)
loginWebView.loadRequest(request)
```

要想让 Web View 加载页面，用 loadRequest 方法就可以了。

Build 一下，在 WebView 中登录后授权页面就加载了，可是如何在 WebView 中获取到授权码，然后再发起获取 token 的请求呢？ UIWebViewDelegate 中的 webView(_ webView: UIWebView, shouldStartLoadWith request: URLRequest, navigationType: UIWebViewNavigationType) -> Bool 可以做这件事。这个 API 的作用是 webView 每次发请求之前，都要完成当前所有定义的逻辑。其中 request 参数每次都要发送请求地址，那我们先了解这些地址都是什么吧。

在 LoginViewController.swift 最后加入如下代码：

```
extension LoginViewController: UIWebViewDelegate {
  func webView(_ webView: UIWebView, shouldStartLoadWith request: URLRequest, navigationType: UIWebViewNavigationType) -> Bool {
    let absoluteURL = request.url!.absoluteString

    print(absoluteURL)

    return true
  }
}
```

Build 一下，登录后单击 Web View 中绿色的“Authorize”，打开 Debug 区域，看到打印出来的结果是这样的（你看到的和下图结果会有所区别，因为返回的值是不定的）：

```
https://unsplash.com/oauth/authorize?
client_id=a1a50a27313d9bba143953469e415c24fc1096aea3be010bd46d4bd252a60896&response_type=code&redirect_uri=Oslo://
photos&scope=public+read_user+write_likes
https://unsplash.com/oauth/authorize
oslo://photos?code=aadcea3a3fbf173a48f753e50068dd0343550d923dd80f03065a9e326fddf086
```

从上图看到，小小的一个单击授权的操作，竟然发送了三个请求。

第一个 https://unsplash.com/oauth/authorize?client_id=a1a50a27313d9bba143953469e415c24fc1096aea3be010bd46d4bd252a60896&response_type=code&redirect_uri=Oslo://photos&scope=public+read_user+write_likes 自然是我们在 func viewDidLoad() 中定义的要加载授权页面的请求。

第二个 https://unsplash.com/oauth/authorize 是服务端获得请求后的一次跳转。

第三个 oslo://photos?code=aadcea3a3fbf173a48f753e50068dd0343550d923dd80f03065a9e326fddf086 是包含用来获得 token 的 code 的请求。

接下来就需要将 oslo://photos?code=aadcea3a3fbf173a48f753e50068dd0343550d923dd80f03065a9e326fddf086 处理一下，因为只需要 code 的部分。然后用 code 发送获取 token 的请求。将 print(absoluteURL) 一行替换为：

```
if absoluteURL.contains("oslo://photos") {
  let code = absoluteURL.components(separatedBy: "=")[1]

  NetworkService.request(url: URL(string: Constants.Base.UnsplashURL + Constants.Base.Token)!, method: NetworkService.HTTPMethod.POST, parameters: [
    Constants.Parameters.ClientID as Dictionary<String, AnyObject>,
    Constants.Parameters.ClientSecret as Dictionary<String, AnyObject>,
    Constants.Parameters.RedirectURI as Dictionary<String, AnyObject>,
    ["code": code as AnyObject],
    Constants.Parameters.GrantType as Dictionary<String, AnyObject>
  ]) { jsonData in
    print(jsonData)
  }

  return false
}
```

这里我们先获取 code，然后遵照 OAuth 2 的流程，将 code 放进请求参数中，与其他参数一起发送请求。注意，这里的请求是 POST 请求，因为其中包含了太多

敏感信息，POST 请求会将这些信息隐藏。

发送请求后返回的数据为 JSON 数据，打印结果如下：

```
{
    "access_token" = d91df91d02b671d1ff974b741223a27343880a71c6b73937009f55d98639cead;
    "created_at" = 1480131603;
    "refresh_token" = 506c53513981bfe3dd696c8ad34d92a0386de9c76eadc214bbed3c5a4b21075b;
    scope = "public read_user write_likes";
    "token_type" = bearer;
}
```

渴望的 access_token 出现了！拥有 token 用户就可以登录，请求也就结束，因此最后返回了 false。

到此这个界面的任务就完成了，在 return false 上方加入

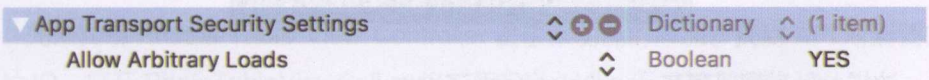
```
self.dismiss(animated: true)
```

返回到 Photos View Controller。

既然明白如何用 Web View 加载网页，也就能把 Profile View Controller 中的加载个人网站页面完成。在 class 内部最后添加：

```
override func viewDidLoad() {
  super.viewDidLoad()

  portfolioWebView.delegate = self

  let requestURL = URL(string: portfolioURL)!
  let request = URLRequest(url: requestURL)
  portfolioWebView.loadRequest(request)
}
```

若发现有些网页并不能加载。那是因为这些网页的协议是 http，不是安全的 https。因此需要通过 App Security 的设置让 http 的访问放行。进入 Info.plist，添加如下图所示内容。



再 Build 一下，可以正常显示了吧！

- UserDefaults -

现在应用的数据都是结束应用后就没有了，但我们辛辛苦苦获得的 token 要一直保留，这样用户才不用每次打开应用都需要重新登录一次。

之前我们保存的数据都是用 NSCache，这仅限于临时的数据存取，下次打开应用还是会消失，因此要寻求其他办法。像之前所说，iOS API 中有很多类都可以保存数据，对于量级比较小、存储结构又简单的数据，我们就用 UserDefaults 吧，听这个名字也可以认为是保存用户默认信息。UserDefaults 和 NSCache 差不多，都是以 key - value 形式储存的。

进入 OperationService.swift，在最下方加入如下代码来定义 token 的读取、保存和删除：

```
struct Token {
  static let userDefaults = UserDefaults.standard

  static func getToken() -> String? {
    return userDefaults.string(forKey: "token")
  }

  static func saveToken(_ token: String) {
    userDefaults.set(token, forKey: "token")
  }

  static func removeToken() {
    userDefaults.removeObject(forKey: "token")
  }
}
```

这里定义了一个单例（Singleton），这样无论在哪里都可以进行 token 的几个操作了。

回到 LoginViewController.swift 中，在 self.dismiss(animated: true) 上方添加：

```
Token.saveToken(accessToken)
```

这样 token 就保存在 userDefaults 中了。

为了区分登录和非登录状态，在 PhotosTableViewController.swift 中 func viewDidLoad() 上方加入如下代码：

```
override func viewWillAppear(_ animated: Bool) {
  super.viewWillAppear(animated)

  if Token.getToken() != nil {
    userBarButton.tintColor = UIColor.colorWithRGB(red: 255, green: 213, blue: 40, alpha: 1.0)
  }
}
```

func viewWillAppear() 会在每次视图显示之前调用。检查 token 是否存在，如果存在，登录的图标会改变颜色。

既然已经登录了，单击登录图标也就不会到授权页面，因此需要将 func userBarButtonDidPressed(_ sender: Any) 内容改为：

```
if Token.getToken() == nil {
  let loginViewController = storyboard?.instantiateViewController(withIdentifier: "LoginViewController") as! LoginViewController
  present(loginViewController, animated: true)
} else {
  let meViewController = storyboard?.instantiateViewController(withIdentifier: "MeViewController") as! MeViewController
  show(meViewController, sender: sender)
}
```

有了 token 后，单击登录图标就进入了 Me View Controller 中了。

- POST -

在发送获取 token 的请求时，曾提到需要用 POST 方法来发送请求，这样敏感信息就不会外泄。由于 token 是用户的标识，有了 token 就可能影响用户的数据，所以带有 token 的请求信息都应该被保护好，因此用 POST 方法来发送。

添加 / 取消喜欢的图片就是一个案例，下面我们就用 POST 方法为喜欢的照片做添加 / 取消。

进入 https://unsplash.com/documentation#like-a-photo 来观察一下 API。很简单，只要带上一个图片的 ID，并在参数上加 Authorization: Bearer ACCESS_TOKEN 就完成了。

先来实现 Photos Table View Controller 里喜欢功能的添加。找到 API 所需要的图片 ID。打开 PhotoTableViewCell.swift，在定义 isLike 下方加入接收 ID：

```
var photoID: String = ""
```

进入 PhotosTableViewController.swift，在 func tableView(_tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell 的 return cell 上方加入：

```
cell.photoID = photo.id
```

将数据传递给 cell 的 photoID。接到数据后，就可以发送 API 请求。将 func heartButtonDidPressed(_ sender: Any) 内容改为：

```
if let token = Token.getToken() {
  let url = URL(string: Constants.Base.UnsplashAPI + "/photos/" + photoID + "/like")!

  if !isLike {
    isLike = !isLike
    heartCountLabel.text = "\(Int(heartCountLabel.text!)! + 1)"

    NetworkService.request(url: url, method: NetworkService.HTTPMethod.POST, headers: ["Authorization": "Bearer " + token])
  } else {
    isLike = !isLike
    heartCountLabel.text = "\(Int(heartCountLabel.text!)! - 1)"

    NetworkService.request(url: url, method: NetworkService.HTTPMethod.DELETE, headers: ["Authorization": "Bearer " + token])
  }
} else {

}
```

如果用户已经登录，就可以添加喜欢的数据信息。将 URL 设置为 API 要求的格式，如果用户还没有添加喜欢的图片，则发送添加喜欢图片的 API 请求。如果已经添加了的图片喜欢，则取消请求，取消的 API 在 https://unsplash.com/documentation#unlike-a-photo 一样要写清楚，只需要将请求方法由 POST 改为 DELETE 就可以了。

这是用户已经登录的情况，那么没有登录的情况呢？需要提示用户去登录，即要显示登录界面。可登录界面是由 View Controller 执行，不是 Cell 执行，怎么办？还是用 delegate 吧。在 PhotoTableViewCellDelegate 的 protocol 里加入：

```
func heartButtonDidPressed(_ sender: Any)
```

再补全刚才的 else 内容：

```
delegate?.heartButtonDidPressed(sender)
```

然后在 PhotosTableViewController.swift 中定义它，在遵循 PhotoTableViewCellDelegate 的 extension 中加入：

```
func heartButtonDidPressed(_ sender: Any) {
  let loginViewController = storyboard?.instantiateViewController(withIdentifier: "LoginViewController") as! LoginViewController
  present(loginViewController, animated: true)
}
```

先从 storyboard 中得到 Login View Controller，然后显示。

为图片加上红心，再去 https://unsplash.com/@ 你的用户名 /likes 下查看一下，数据是否已经生效。

下面来实现 Personal Photo View Controller 中喜欢功能的添加。方法是一样的，在 func heartButtonDidPressed(_ sender: Any) 中添加：

```
if let token = Token.getToken() {
  let url = URL(string: Constants.Base.UnsplashAPI + "/photos/" + photo.id + "/like")!

  if !photo.isLike {
    heartButton.setBackgroundImage(♥heart-liked, for: .normal)
    heartCountLabel.text = "\(Int(heartCountLabel.text!)! + 1)"

    NetworkService.request(url: url, method: NetworkService.HTTPMethod.POST, headers: ["Authorization": "Bearer " + token])
  } else {
    heartButton.setBackgroundImage(♡heart-outline, for: .normal)
    heartCountLabel.text = "\(Int(heartCountLabel.text!)! - 1)"

    NetworkService.request(url: url, method: NetworkService.HTTPMethod.DELETE, headers: ["Authorization": "Bearer " + token])
  }
} else {
  let loginViewController = storyboard?.instantiateViewController(withIdentifier: "LoginViewController") as! LoginViewController
  present(loginViewController, animated: true)
}
```

唯一不同的是显示 Login View Controller 没有通过 delegate，因为这个 View Controller 自己就可以完成。

Build 一下，是不是这里也成功了呢?

- 用 delegate 来传输数据 -

从 Storyboard 上看，Photos Table View Controller、Profile View Controller、Published Photos Collection View Controller 和 Liked Photos Collection View Controller 都可以到 Personal Photo View Controller，但它们之间添加 / 取消喜欢的数据却是不同步的。

之前介绍了用 Segue 来传输 View Controller 之间的数据，可在有些情况下 Segue 不是万能的。比如 Segue 在数据还没准备好之前就被触发了怎么办? 从 A 到 B 触发 Segue，但是从 B 到 A 触发不了该怎么办? 不过 delegate 可以做到 View Controller 之间“通知”和“传输”的作用，那就用它来试试数据的传输吧!

进入 PersonalPhotoViewController.swift，在 class 外部顶端添加如下代码：

```
protocol PersonalPhotoViewControllerDelegate: class {
  func heartButtonDidPressed(with photoID: String, isLike: Bool, heartCount: Int)
}
```

这里要将图片的 ID、是否添加了喜欢以及喜欢的数量传过去。

在 class 内部 IBOutlet 的上方添加：

```
weak var delegate: PersonalPhotoViewControllerDelegate?
```

delegate 定义好后，就确认什么时候触发它。当然是在单击红心按钮的时候触发，找到 func heartButtonDidPressed(_ sender: Any)，在 if 和 else 两个发送请求的代码之前加入：

```
delegate?.heartButtonDidPressed(with: photo.id, isLike: photo.isLike, heartCount: photo.heartCount)
```

这样在 PersonalPhotoViewController.swift 中的 delegate 所用到的方法、变量的定义和触发时机都定义完成，接着定义方法的机制。切换到 PhotosTableViewController.swift 中，在最下方添加：

```
extension PhotosTableViewController: PersonalPhotoViewControllerDelegate {
  func heartButtonDidPressed(with photoID: String, isLike: Bool, heartCount: Int) {
    if let indexPathRow = photos.index(where: { $0.id == photoID }) {
      photos[indexPathRow].isLike = isLike
      photos[indexPathRow].heartCount = heartCount

      let indexPath = IndexPath(row: indexPathRow, section: 0)

      tableView.reloadRows(at: [indexPath], with: .automatic)
    }
  }
}
```

这里定义了 delegate 的方法。首先把传过来的图片 ID 在当前的 View Controller 数据中做匹配，找到符合这个 ID 的图片，并获得它的位置。然后根据所在位置，将这张图片的是否喜欢和喜欢的数量重新赋值。最后刷新这个位置的 cell。

在 Personal Photo View Controller 加个喜欢的 Photos Table View Controller 同步应该没问题了！不过如果想做到从任何界面进入 Personal Photo View Controller，数据都同步到 Photos Table View Controller 中，就需要在合适的时机都对 delegate 赋值。比如用户在 Photos Table View Controller 上单击用户头像，跳转到 Profile View Controller 中，再从这里到了 Personal Photo View Controller，那么 Photos Table View Controller 中的 destinationViewController.delegate = self 就不会生效，delegate 自然也不会传输数据。

在 ProfileViewController.swift、PublishedPhotosCollectionViewController.swift 和 LikedPhotosCollectionViewController.swift 中，找到 func prepare(for segue: UIStoryboardSegue, sender: Any?)，在其中的 if segue.identifier == “PhotoSegue" 内部最下方加入：

```
if let photosTableViewController = navigationController?.viewControllers[0] as? PhotosTableViewController {
  destinationViewController.delegate = photosTableViewController
}
```

这样就将 Personal Photo View Controller 的 delegate 赋给 Personal Photo View Controller。完成后，任何界面进入 Personal Photo View Controller，数据都会同步到 Photos Table View Controller 中。

懂得了 delegate 的数据传输原理，就可以把 Me View Controller 中 Container View 的问题解决。Me View Controller 通过 Container View 连接了两个 Collection View，连接的方法也是通过 Segue。这个 Segue 会在加载 Me View Controller 后迅速触发，而只有在 Collection View 所需要的 API 数据准备好时，Collection View 才可以调用 Load 方法。这里就需要用 delegate 来解决遇到的问题了。

进入 MeViewController.swift，在顶部添加：

```
protocol PassDataDelegate: class {
  func pass(userName: String, photosCount: Int)
}
```

在 class 内部 IBOutlet 上方添加：

```
weak var publishedPhotosdelegate: PassDataDelegate?
weak var likedPhotosdelegate: PassDataDelegate?
```

因为要在 userName、publishedTotalCount 和 likedTotalCount 这三个数据都准备好时才传输数据，所以在 if let userName = jsonData["username"] as? String 内部最下方添加：

```
self.publishedPhotosdelegate?.pass(userName: userName, photosCount: self.publishedTotalCount)
self.likedPhotosdelegate?.pass(userName: userName, photosCount: self.likedTotalCount)
```

设置好 delegate 后给 delegate 赋值，在 class 内部添加：

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
  if segue.identifier == "PublishedPhotosSegue" {
    if let destinationViewController = segue.destination as? PublishedPhotosCollectionViewController {
      publishedPhotosdelegate = destinationViewController
    }
  } else if segue.identifier == "LikedPhotosSegue" {
    if let destinationViewController = segue.destination as? LikedPhotosCollectionViewController {
      likedPhotosdelegate = destinationViewController
    }
  }
}
```

切换到 PublishedPhotosCollectionViewController.swift 中，在最下方添加：

```
extension LikedPhotosCollectionViewController: PassDataDelegate {
  func pass(userName: String, photosCount: Int) {
    self.userName = userName
    self.likedTotalCount = photosCount
  }
}
```

有了 userName，就可以判断发送请求的时机了，在 class 内部最下方添加：

```
func retryLoad() {
  if userName != "" {
    load()

    return
  } else {
    delay(1.0) {
      self.retryLoad()
    }
  }
}
```

如果 userName 不为空，就发送请求，否则延迟 1 秒重试。

最后在 func viewDidLoad() 最下方添加：

```
retryLoad()
```

切换到 LikedPhotosCollectionViewController.swift 中，用同样的方法，在最下方添加：

```
extension PublishedPhotosCollectionViewController: PassDataDelegate {
  func pass(userName: String, photosCount: Int) {
    self.userName = userName
    self.publishedTotalCount = photosCount
  }
}
```

在 class 内部最下方添加：

```
func retryLoad() {
  if userName != "" {
    load()

    return
  } else {
    delay(1.0) {
      self.retryLoad()
    }
  }
}
```

在 func viewDidLoad() 最下方添加：

```
retryLoad()
```

Build 一下，图片就正常显示出来了！

根据设计，不同的设备屏幕尺寸，cell 应该有不同的大小，而 storyboard 中无法定义 cell 的 autolayout，所以这里我们可以通过代码来实现。要想让 cell 获得屏幕尺寸，同样可以用 delegate。在 protocol PassDataDelegate 内部添加：

```
func pass(width: CGFloat)
```

将屏幕宽度传输给其他 View Controller。

在 func viewDidLoad() 中 super.viewDidLoad() 下方添加：

```
likedPhotosdelegate?.pass(width: self.view.frame.width * 0.56)
publishedPhotosdelegate?.pass(width: self.view.frame.width * 0.56)
```

切换到 PublishedPhotosCollectionViewController.swift 中，在最下方 extension PublishedPhotosCollectionViewController: PassDataDelegate 处添加：

```
func pass(width: CGFloat) {
  self.width = width
}
```

同样的方法，在 LikedPhotosCollectionViewController.swift 最下方 extension PublishedPhotosCollectionViewController: PassDataDelegate 处添加：

```
func pass(width: CGFloat) {
  self.width = width
}
```

```
@IBOutlet var segmentedControl: UISegmentedControl!
@IBOutlet var profileImageView: UIImageView!
@IBOutlet var nameLabel: UILabel!
@IBOutlet var bioLabel: UILabel!
@IBOutlet var publishedPhotoContainerView: UIView!
@IBOutlet var likedPhotoContainerView: UIView!
```

在不同设备模拟器上 Build 一下，cell 的尺寸也一样将随屏幕大小而不同了！

现在我们的应用还差最后一个界面没有完成，就是 Me 界面。这个界面综合了前面的大部分知识。本书只对特殊部分做出说明。源代码可以查看“设计资源和源代码”一章的索取方式。

建立 MeViewController.swift，关联到 Storyboard 对应 View Controller，连接组件到代码：

```
@IBOutlet var segmentedControl: UISegmentedControl!
@IBOutlet var profileImageView: UIImageView!
@IBOutlet var nameLabel: UILabel!
@IBOutlet var bioLabel: UILabel!
@IBOutlet var publishedPhotoContainerView: UIView!
@IBOutlet var likedPhotoContainerView: UIView!
```

```
@IBAction func logoutButtonDidPressed(_ sender: Any) {
  let alertViewController = UIAlertController(title: "Logout", message: "Logout intro", preferredStyle: .alert)
  let confirm = UIAlertAction(title: "Logout action title", style: .default) { action in
    Token.removeToken()

    _ = self.navigationController?.popToRootViewController(animated: true)
  }
  let cancel = UIAlertAction(title: "Logout cancel title", style: .cancel)

  alertViewController.addAction(confirm)
  alertViewController.addAction(cancel)

  present(alertViewController, animated: true)
}
```

定义开关键单击后的登出账号行为：

```
@IBAction func logoutButtonDidPressed(_ sender: Any) {
  let alertViewController = UIAlertController(title: "Logout", message: "Logout intro", preferredStyle: .alert)
  let confirm = UIAlertAction(title: "Logout action title", style: .default) { action in
    Token.removeToken()

    _ = self.navigationController?.popToRootViewController(animated: true)
  }
  let cancel = UIAlertAction(title: "Logout cancel title", style: .cancel)

  alertViewController.addAction(confirm)
  alertViewController.addAction(cancel)

  present(alertViewController, animated: true)
}
```

在 class 内部最下方添加：

```
private func generateSegmentedControlImage() -> UIImage {
  let rectangle = CGRect(x: 0, y: 0, width: 1, height: segmentedControl.frame.size.height)
  UIGraphicsBeginImageContext(rectangle.size)

  if let ctx = UIGraphicsGetCurrentContext() {
    ctx.setFillColor(UIColor.white.cgColor)
    ctx.addRect(rectangle)
  }

  let image = UIGraphicsGetImageFromCurrentImageContext()!
  UIGraphicsEndImageContext()

  return image
}
```

这个方法生成了一个宽度为 1，高度为 Segmented Control 的白色图片。之后用作自定义 Segmented Control 样式。

在这个方法下面继续添加：

```
private func generateSelectedBorder(under view: UIView) {
  if segmentedControl.selectedSegmentIndex == 0 {
    let bottomBorder = CALayer()
    bottomBorder.name = "bottomBorder"
    bottomBorder.frame = CGRect(x: -view.center.x / 2.5, y: view.frame.size.height + 2, width: 13, height: 1)
    bottomBorder.backgroundColor = UIColor.colorWithRGB(red: 95, green: 95, blue: 95, alpha: 1.0).cgColor
    view.layer.addSublayer(bottomBorder)
  } else {
    let bottomBorder = CALayer()
    bottomBorder.name = "bottomBorder"
    bottomBorder.frame = CGRect(x: view.center.x * 3, y: view.frame.size.height + 2, width: 13, height: 1)
    bottomBorder.backgroundColor = UIColor.colorWithRGB(red: 95, green: 95, blue: 95, alpha: 1.0).cgColor
    view.layer.addSublayer(bottomBorder)
  }
}
```

这个方法生成了一个“横条”，用来提示当前在 segment 下和设计图是一致的。

有了这两个方法，就可以定义 Segmented Control 了。在 class 内部添加：

```
override func viewDidLoad() {
  super.viewDidLoad()

  segmentedControl.setDividerImage(generateSegmentedControlImage(), forLeftSegmentState: .normal, rightSegmentState: .normal, barMetrics: .default)
  segmentedControl.setBackgroundImage(generateSegmentedControlImage(), for: .normal, barMetrics: .default)
  segmentedControl.setTitleTextAttributes([NSForegroundColorAttributeName: UIColor.colorWithRGB(red: 155, green: 155, blue: 155, alpha: 1.0)], for: .normal)
  segmentedControl.setTitleTextAttributes([NSForegroundColorAttributeName: UIColor.colorWithRGB(red: 92, green: 92, blue: 92, alpha: 1.0)], for: .selected)

  generateSelectedBorder(under: segmentedControl.subviews[0])
}
```

再定义 Segmented Control 的行为：

```
@IBAction func segmentedControlDidChange(_ sender: Any) {
  for subview in segmentedControl.subviews {
    if let bottomBorderLayer = subview.layer.sublayers?.filter({ $0.name == "bottomBorder" }) {
      for layer in bottomBorderLayer {
        layer.removeFromSuperlayer()
      }
    }
  }

  if segmentedControl.selectedSegmentIndex == 0 {
    generateSelectedBorder(under: segmentedControl.subviews[0])

    publishedPhotoContainerView.isHidden = false
    likedPhotoContainerView.isHidden = true
  } else {
    generateSelectedBorder(under: segmentedControl.subviews[1])

    publishedPhotoContainerView.isHidden = true
    likedPhotoContainerView.isHidden = false
  }
}
```

Me 界面的特殊地方是，它存在两个 Container View，这两个 Container View 又分别连接了两个 Collection View Controller。当界面一加载就会触发 Container View 与 Collection View Controller 连接的 Segue，导致有些数据还没有准备好，无法通过 Segue 来传输。因此同样需要用 delegate 来完成数据传输。

其他

让一款应用具有吸引力，只具有简单的实现功能还不够。需要视觉的美化、多语言的支持、更多辅助性功能等这样才能让应用变得更具魅力。

接下来的章节会介绍到 iOS 的动画和本地化语言。有动画技术方面 Apple 一直在完善相应接口，让开发者可以更加自如地控制动画的每一个阶段，想必 Apple 本身也认为动画对于用户的体验有至关重要的影响，不但能让应用有趣，还能帮助用户理解页面之间的关系和开发者的表达方式，比如我们熟悉的 func present(_ viewControllerToPresent: UIViewController, animated flag: Bool, completion: (() -> Swift.Void)? = nil) 就是个好案例。

对于本地化语言，就不必多说了，看得懂的应用，才能使用。

- 动画 -

制作 iOS 动画有很多种方法，Apple 提供的 API 也多种多样。本书只介绍比较好理解的 UIView 的动画，来感受一下其中的原理。

要想做动画，首先要了解动画的本质。动画是由一帧一帧的图像快速交替形成的。好比一个本子，每页都画一张图，在快速翻看的时候，就会觉得这些图“动”了起来。所谓的一帧，可以理解为一张图，只要一秒钟不低于 24 帧，人眼就会产生“动”起来的错觉。在 iOS 上开发，也是这样一帧一帧绘制出来的，不过如此复杂的工作，Apple 已经为我们做好了，我们只需要用现成的 API 来调用即可。

这每一帧图像的位置、大小、透明度、颜色等组成了动画，那么在什么阶段，让这些属性如何变化，才是我们定义动画的关键。

估计你已经想到第一个单击后要做的动画是什么了——就是 Loading！现在打开应用一片空白，如果用户网络不好单击后可能会以为出现了问题而直接关掉应用，还是马上加一个 Loading 的动画让用户多一些耐心吧。

在 Views 文件夹内建立一个名为 LoadingView.swift 的文件。我们将在这里存储制作好的 Loading View，想用的时候加上它就可以了。

首先在内部加入如下代码：

```
import UIKit

class LoadingView: UIView {
  private let images = [mountain, kaminarimon_gate, tram, blitzcam]
  private var imageView: UIImageView! {
    didSet {
      imageView.contentMode = .scaleAspectFit
      imageView.frame = CGRect(x: self.frame.size.width / 2 - 20,
                               y: self.frame.size.height / 2 - 20,
                               width: 40,
                               height: 40)
    }
  }
  private var count: Int = 0
  private let transitions: [UIViewAnimationOptions] = [.transitionFlipFromLeft,
                                                       .transitionFlipFromRight,
                                                       .transitionFlipFromTop,
                                                       .transitionFlipFromBottom]
}
```

这里先定义了需要交替的几张图片，然后定义了显示他们的 Image View。count 代表轮换的次数。transitions 中定义所需要的几种动画翻转方式。

继续在下方添加：

```
override init(frame: CGRect) {
  super.init(frame: frame)

  backgroundColor = UIColor.white
}

required init?(coder aDecoder: NSCoder) {
  fatalError("init(coder:) has not been implemented")
}
```

初始化 Loading View 的方法。定义尺寸和背景颜色。

继续在下方添加：

```
private func loopImage() {
  if count < images.count {
    UIView.transition(with: self.imageView,
                      duration: 0.5,
                      options: [transitions.randomItem(), .curveEaseInOut],
                      animations: {
      self.imageView.image = self.images[self.count]
    }, completion: { _ in
      delay(0.5) {
        self.count += 1
        self.loopImage()
      }
    })
  } else {
    count = 0
    loopImage()
  }
}
```

这里先交替显示 4 张图，交替的方式为随机一种 transition。时间函数是 curveEaseInOut。在 iOS 中，时间函数简单分为以下 4 种，如下图所示。

函数名称	效果
curveLinear	快慢不变
curveEaseOut	慢出
curveEaseIn	慢入
curveEaseInOut	慢入慢出

之所以定义这些，是因为这样的效果更符合自然界事物运动的法则，动画看起来会更加自然。图片显示完后，计数器 count 归 0，重新运行这个方法。

定义完动画，就要调用动画了，动画原理是一帧一帧“画”出来的，那么就在 func draw(_ rect: CGRect) 这个 API 中调用动画吧。最后在下方添加：

```
override func draw(_ rect: CGRect) {
  imageView = UIImageView()

  self.addSubview(imageView)

  loopImage()
}
```

这样 Loading View 就做好了，快去需要的地方添加一下吧。打开 PhotosTableViewController.swift，在 fileprivate var 下方添加：

```
private var loadingView: LoadingView! {
  didSet {
    loadingView.frame.origin = self.view.frame.origin
    loadingView.frame.size.width = self.view.frame.size.width
  }
}
```

并在 feedRefreshControl.addSubview(self.loadingView) 上方添加：

```
self.loadingView = LoadingView()
self.loadingView.frame.size.height = feedRefreshControl.frame.size.height
```

这样就把 Loading View 和 Refresh Control 绑在了一起，Build 一下看看吧！

同样在 Profile View Controller 中也可以加入我们写好的 Loading View。进入

ProfileViewController.swift，在 fileprivate var 下方添加：

```
private var loadingView: LoadingView! {
  didSet {
    loadingView.frame = collectionView.bounds
    loadingView.frame.size.width = self.view.frame.size.width
  }
}
```

在 func viewDidLoad() 内部 load 方法上方添加：

```
loadingView = LoadingView()
collectionView.addSubview(loadingView)
```

最后在 load 方法 self.collectionView.reloadData() 下方加入：

```
self.loadingView.removeFromSuperview()
```

加载结束后就删除 Loading View。Build 一下，这样就不怕用户多等一会儿了！

除了 transition，UIView 还提供了一些基础的组件动画 API。现在我们的应用在显示图片的时候是在加载完成后突然出现的，有没有更好的显示方式呢？在浏览一些网页的时候会发现图片是“闪”出来的，这背后的原理就是用了动画。把它分解一下，其实就是在一个很短的时间内，透明度由 0 变到了 1。有了这个认识，就可以实际地操作了。打开 PhotosTableViewController.swift，将 func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) 中 if let photoURL = URL(string: photo.imageURL) 里面的 if let updateCell = tableView.cellForRow(at: indexPath) as? PhotoTableViewCell 的内容替换为：

```
updateCell.photoImageView.alpha = 0

UIView.animate(withDuration: 0.3) {
  updateCell.photoImageView.alpha = 1
  updateCell.photoImageView.image = image
}
```

这里先让 Image View 的透明度为 0，然后用 0.3 秒的时间让透明度再变为 1，同时显示图片。Build 一下看看吧，有动画的效果比没有动画的效果强很多。

同样的方法也适用于其他几个 Collection View，自己试试看吧！

不仅可以让透明度变化，组件的位置也可以动画，而且还可以同时进行。试试让单击下载图片按钮后的文案有一些动画。进入 PersonalPhotoViewController.swift，将 func save(_ image: UIImage, didFinishSavingWithError error: NSError?, contextInfo: UnsafeRawPointer) 中 else 内部的内容改为：

```
savePhotoLabel.text = "\(emoji.randomItem())" + " Photo is saved to Album."
savePhotoLabel.transform = CGAffineTransform(translationX: -20, y: -20)

UIView.animate(withDuration: 1, animations: {
  self.savePhotoLabel.alpha = 1
  self.savePhotoLabel.transform = CGAffineTransform.identity
}, completion: { _ in
  delay(1) {
    UIView.animate(withDuration: 1, animations: {
      self.savePhotoLabel.alpha = 0
      self.savePhotoLabel.transform = CGAffineTransform(translationX: 20, y: 20)
    })
  }
})
```

这里先让 savePhotoLabel 的位置往左上移动 20，然后开始动画 1 秒，透明度为 1，同时回到原来位置。这个动画完成后，停留 1 秒，继续下一个动画，为时 1 秒，位置向右下方移动 20。

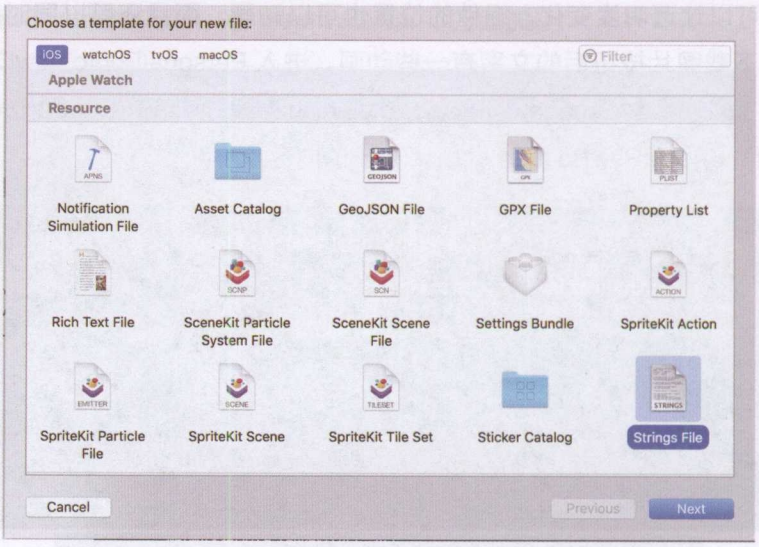
单击下载图片的按钮，这样的动画更加深了“图片已经保存到相册”的印象。

iOS 还有很多可以定义动画的方式，比如在 layer 上定义 keyframes，更精细地操作整个动画过程，以及 View Controller 之间的 transition 动画过渡等，如果细细说来可以单独写上一本书。这么富有创造性的部分，就由各位发挥想象尽情尝试吧！

- 本地化语言 -

想做本地化语言，首先要将分散在各个文件中的应用文案集中起来，然后通过 key - value 的形式来访问。比如一个名为 Hello 的 key，可以对应两个赋值，分别为 Hello 和你好，这样在需要文案的地方读取 Hello 这个 key 就可以实现多语言了。

Xcode 提供了一个集中文案的好地方——在 Helper 文件夹中创建新文件，选择 iOS - Resource - StringsFile。



命名为 Localizable.strings，注意这里不要用其他的命名，否则每次调用 NSLocalizedString(_ key: String, tableName: String? = default, bundle: Bundle = default, value: String = default, comment: String) 时都需要指定名称，这个方法后面会运用。

在 Localizable.strings 中添加以下内容：

```
// MARK: Photos Table View Controller
"Feature" = "Feature";

// MARK: Profile View Controller
"No Location" = "No Location";
"Website" = "Website";
"%@'s website" = "%@'s website";


// MARK: Personal Photo View Controller
"Save photo failed" = "⚠️ Photo is not saved. You may check the photo permission or storage.";
"Photo saved" = " Photo is saved to Album.";


// MARK: Exif View
"Published" = "Published";
"Dimensions" = "Dimensions";
"Camera Make" = "Camera Make";
"Camera Model" = "Camera Model";
"Aperture" = "Aperture";
"Exposure Time" = "Exposure Time";
"Focal Length" = "Focal Length";
"ISO" = "ISO";


// MARK: Statistics View
"Downloads" = "Downloads";
"Views" = "Views";
"Likes" = "Likes";


// MARK: Login View Controller
"Back" = "Back";


// MARK: Me View Controller
"Logout" = "Logout Unsplash";
"Logout intro" = "After logout you could not access your personal photos and specific functions like ❤️ a photo.";
"Logout action title" = "Logout";
"Logout cancel title" = "Cancel";
"Published count" = "%@ Published";
"Liked count" = "%@ Liked";
```

“=”左边的是 key，右边的是 value。在 Misc.swift 最下方添加如下代码：

```
public func localize(with key: String) -> String {
  return NSLocalizedString(key, comment: "")
}

public func localizedFormat(with key: String, and argument: String) -> String {
  return String(format: localize(with: key), argument)
}
```

这样就不用每次调用 NSLocalizedString(_ key: String, tableName: String? = default, bundle: Bundle = default, value: String = default, comment: String) 时都要带着 Comment 参数了，这个参数只起说明作用，自己能看明白不用也可以。

实验一下，将 storyboard 中 Photos Table View Controller 上导航栏的“Feature”删除，然后在 PhotosTableViewController.swift 中 func viewDidLoad() 里面的 super.viewDidLoad() 下方加入：

```
navigationItem.title = localize(with: "Feature")
```

导航栏显示出来 Feature 对应的值，不过现在还只是英文的。

依此类推，把应用其他地方的文案都替换成 localizedString。

接下来就开始真正的多语言部分了。单击 Navigator 中的蓝色图标 Oslo - PROJECT 中蓝色图标的 Oslo - Info - Localizations - + - Chinese(Simplified) (zh-Hans)。之后单击 Localizable.strings 文件，单击 Utilities - Localize··· - Base。

Base 的意思是无论什么语言都用当前定义的英文。单击 Localize 后，再点选 Chinese(Simpl··· ，然后进入 Localizable.strings (Chinese(Simplified))。在这里就可

以定义中文内容了，如下：

```
// MARK: Photos Table View Controller
"Feature" = "精选";

// MARK: Profile View Controller
"No Location" = "没有位置";
"Website" = "个人网站";
"%@'s website" = "%@ 的个人网站";


// MARK: Personal Photo View Controller
"Save photo failed" = "⚠️ 图片没有保存。请检查系统隐私设置或是否存储已满。";
"Photo saved" = " 图片已保存到相册中。";


// MARK: Exif View
"Published" = "发布时间";
"Dimensions" = "尺寸";
"Camera Make" = "相机品牌";
"Camera Model" = "相机型号";
"Aperture" = "光圈";
"Exposure Time" = "曝光时间";
"Focal Length" = "焦距";
"ISO" = "ISO";

// MARK: Statistics View
"Downloads" = "下载";
"Views" = "浏览";
"Likes" = "喜欢";


// MARK: Login View Controller
"Back" = "返回";


// MARK: Me View Controller
"Logout" = "登出 Unsplash 帐号";
"Logout intro" = "登出后将无法访问个人图片，也无法 ❤️ 喜欢的图片。";
"Logout action title" = "登出";
"Logout cancel title" = "取消";
"Published count" = "%@ 发布";
"Liked count" = "%@ 喜欢";
```

将系统语言切换到简体中文。

还有一个地方没有本地化，就是 info.plist 中的 Privacy - Photo Library Usage Description 内容。用同样的方法创建 InfoPlist.strings，内容如下：

```
"NSPhotoLibraryUsageDescription" = "Oslo would save photo to your album 🌄";
```

中文版内容为：

```
"NSPhotoLibraryUsageDescription" = "Oslo 想要将下载的图片保存到你的相册中 🌄";
```

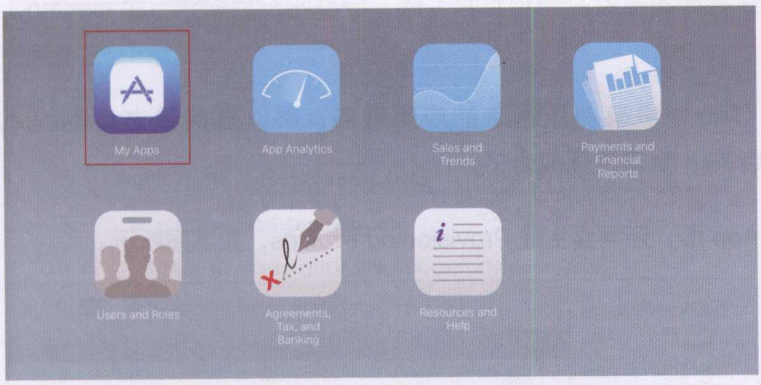
这样就没问题了。

- 提交 TestFlight 测试 -

注意以下操作需要 Apple Developer 账户，并处于付费状态。

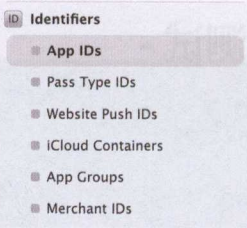
在应用正式提交到 App Store 审核以前，稳妥的做法是邀请一些人帮助在多机型上测试一下，这就要用到 TestFlight。当应用提交到 TestFlight 并且审核通过后，受邀请的人员安装 TestFlight 应用（https://itunes.apple.com/us/app/testflight/id899247664?mt=8）后就可以在里面下载试用了。下面就来了解一下怎么提交应用到 TestFlight 吧！

打开浏览器访问 https://itunesconnect.apple.com/，选择左上角的 My Apps。



单击左上角的 + - New App。

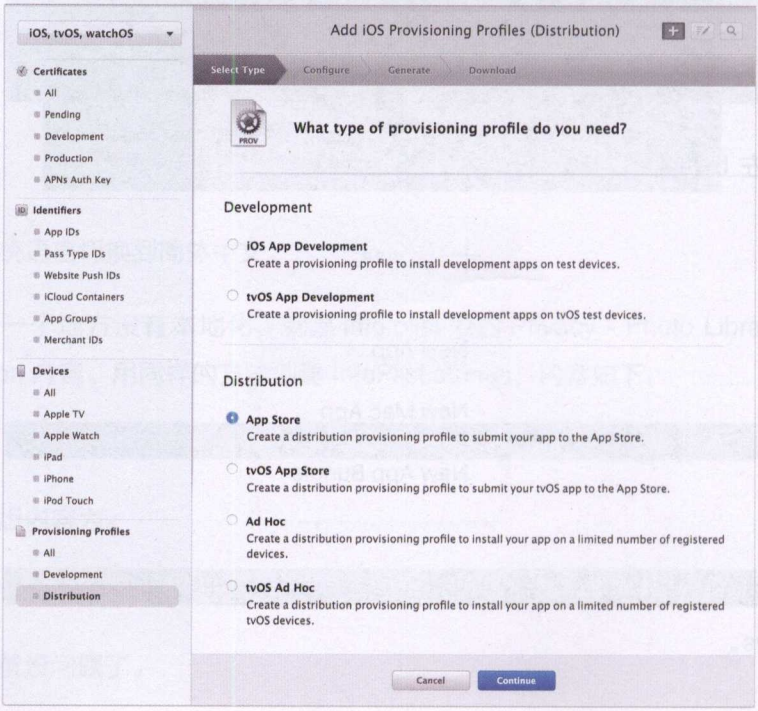
在之后的弹窗中找到 Developer Portal，要在这里先注册 App ID。单击左侧栏的 App IDs。



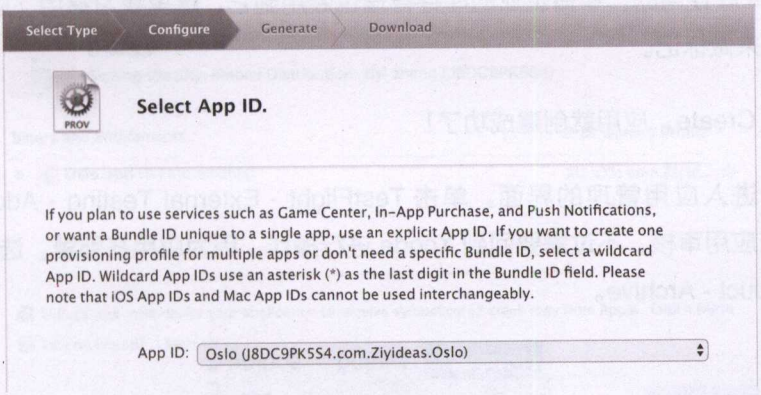
单击右上方 + 之后出现注册 App ID 的表格。App ID Description - Name 一栏填入 Oslo。App ID Suffix 按建议填写为 com.Ziyideas.Oslo，此名称可以和 Xcode 中的 Target 设置保持一致，如下图所示。

由于 Oslo 中没有使用特殊的应用服务，所以直接单击 Continue。再单击 Register 后 App ID 就注册成功了。

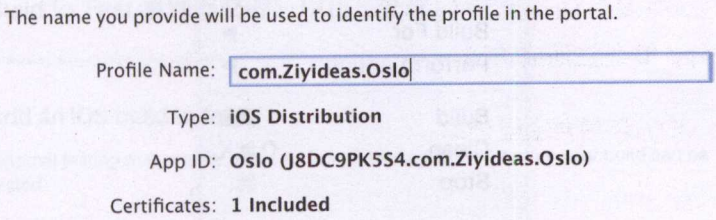
有了 App ID，还需要生成 Provisioning Profiles，如下图所示。



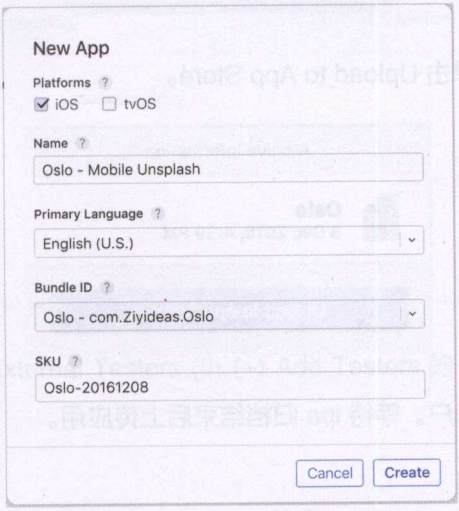
单击 Continue 后，选择刚刚设置的 App ID，如下图所示。



单击 Continue 后，选择自己的账号，为了保证唯一性这里的命名沿用了 App ID，如下图所示。



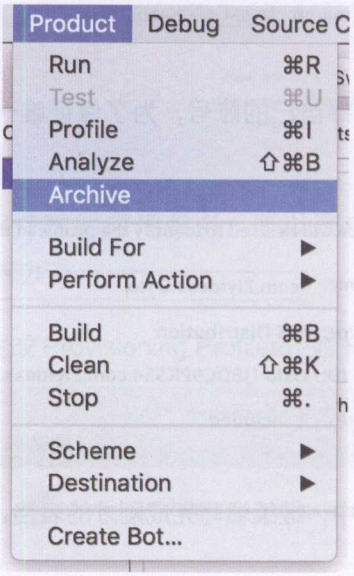
回到 iTunes Connect 中，继续填写完成刚才的表格，如下图所示。



也许你会对 SKU 感到陌生，SKU 的全称为 Stock-keeping Unit，是用来跟踪 App Store 库存用的，零售企业的库存管理也会用到它，这里我习惯用“应用名称 - 年月日”来做标记。

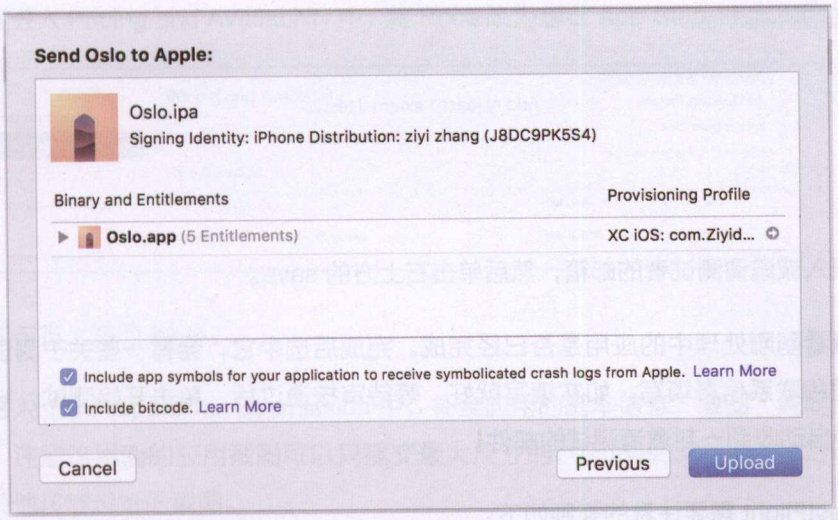
单击 Create，应用就创建成功了！

之后进入应用管理的界面。单击 TestFlight - External Testing - Add Build to Test 提交应用审核，不过需要回到 Xcode 进行操作，因为应用在那里。选择顶部菜单的 Product - Archive。

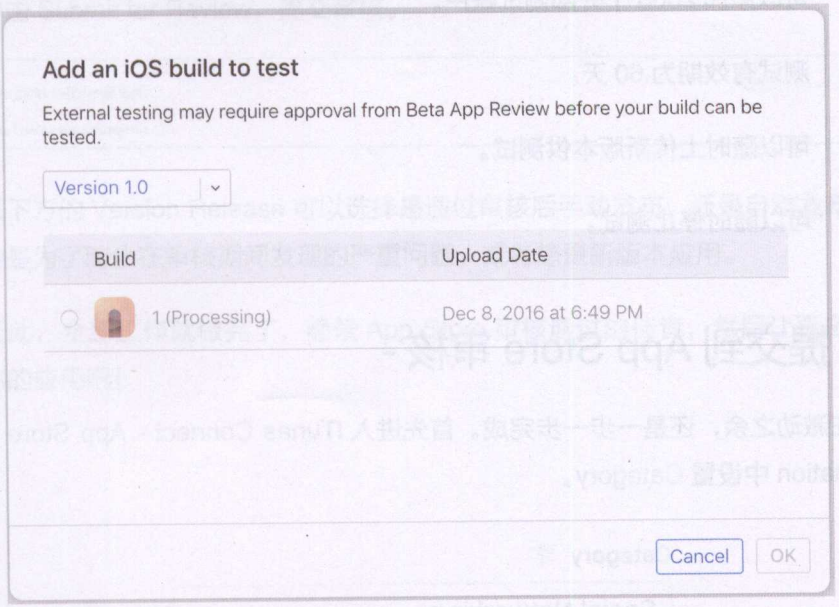


在弹出界面右侧单击 Upload to App Store。

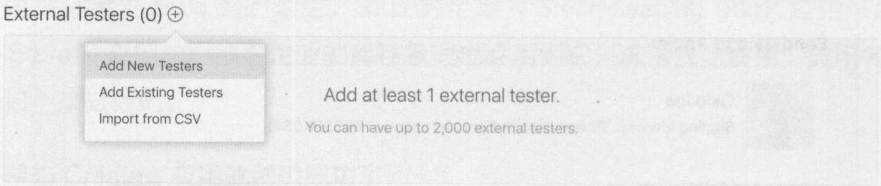
之后登录自己的账户。等待 ipa 归档结束后上传应用。



漫长的等待过后，应用就上传完毕，回到 iTunes Connect 中。刷新网页后再次单击 Add Build to Test 看到提交的应用正在处理。



稍作等待，单击 External Testers (0) (+) Add Testers 的“+”，选择 Add New Testers。



填入被邀请测试者的邮箱，然后单击右上方的 save。

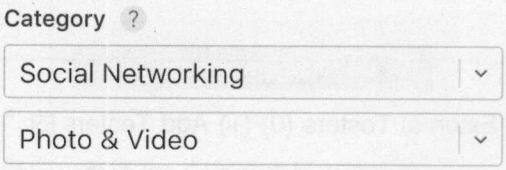
查看刚刚处理中的应用是否已经完成。完成后选中它，会有一些关于测试的应用问题和联系信息填写，如实填写就好。等待审核通过后，单击开始测试，被邀请者就会自动收到一封邀请测试的邮件！

TestFlight 需要注意的规则如下：

- 可以添加 25 个内部测试账户。
- 可以添加 2000 个外部测试账户。
- 测试有效期为 60 天。
- 可以随时上传新版本供测试。
- 可以随时停止测试。

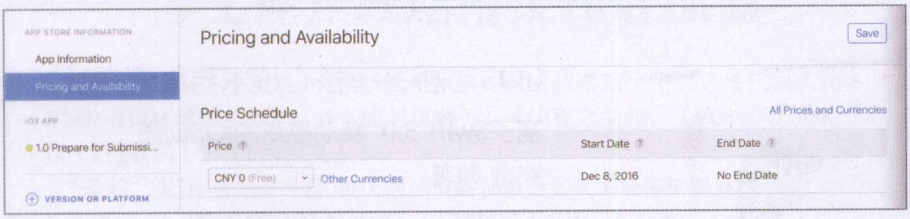
- 提交到 App Store 审核 -

在激动之余，还是一步一步完成。首先进入 iTunes Connect - App Store - App Information 中设置 Category。



单击右上方的 save。

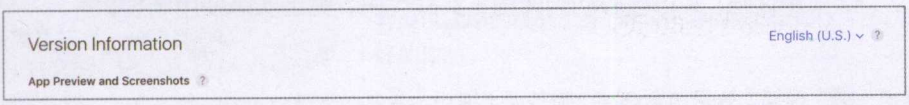
再进入 Pricing and Availability 中，将 Price 改为想在 App Store 中出售的价格。



单击右上方的 save。

进入 1.0 Prepare for Submission 中，补全应用的基本信息、截图、说明、关键词等。注意这里面的应用截图可以只提交最大尺寸的，iTunes Connect 会自动生成对应其他尺寸设备的截图。

在下图部分，可以切换不同的地区，来填写不同的应用信息。比如应用说明，可以填写对应地区的文字，全部填写完成后，单击右上方的 save。确认无误后，就可以单击 Submit for Review，提交审核。



最下方的 Version Release 可以选择是通过审核后手动发布，还是自动发布。这个主要是为了防止在审核期间发现的严重问题，或者想更新版本应用。

至此，全部工作就做完了，静候 App Store 审核通过的佳音，然后让更多的人使用你的应用吧！

相关推荐

《iOS动画——核心技术与案例实战》

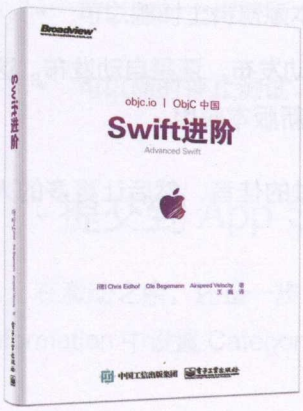
基于Swift 3.0 舞动酷炫的iOS动画

郑微 编著

ISBN 978-7-121-30748-5
2017年2月出版
定价：69.00元
208页

编辑推荐

◎ 层次分明：显示层动画、内容层动画、3D动画、转场动画。
◎ 内容丰富：UIView、Layer、常用转场动画合集。
◎ 适用性强：轻松过滤出适合自己的核心内容。
◎ 实用性强：iOS核心动画架构＋实战代码，贴近实际使用场景。



《Swift进阶》

喵神领衔iOS开发三部曲之完结篇
从低层级编程到高阶抽象完成最终修炼

【德】Chris Eidhof（克里斯·安道夫）
【德】Ole Begemann（奥勒·毕格曼）
【德】Airspeed Velocity（空速网站）著
王巍 译

ISBN 978-7-121-31200-7
2017年5月出版
定价：75.00元
300页

编辑推荐

◎ Swift 非常适于系统编程，同时它也能被用于书写高层级的代码。
◎ 如果你已想深入探索这门语言的奥秘，这是唯一能找到的一本书。
◎ 全球知名 iOS 教学网站 objc，其精品图书更是惠及无数开发者。
◎ 高层级抽象如泛型|协议，低层级如封装C代码、字符串内部实现。

十载耕耘奠定专业地位

博文视点诚邀精锐作者加盟

以书为证彰显卓越品质

《C++Primer（中文版）（第5版）》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目，被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉Zac、李刚、曹江华……

“明星”级作者济济一堂，他们的名字熠熠生辉，与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治，成就**博**古通今、**文**圆质方、**视**角独特、**点**石成金之计算机图书的风向标杆：博文视点。

“凤翱翔于千仞兮，非梧不栖”，博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟，与大师并列于IT专业出版之巅。

英雄帖

江湖风云起，代有才人出。
IT界群雄并起，逐鹿中原。
博文视点诚邀天下技术英豪加入，
指点江山，激扬文字
传播信息技术，分享IT心得

• 专业的作者服务 •

博文视点自成立以来一直专注于IT专业技术图书的出版，拥有丰富的与技术图书作者合作的经验，并参照IT技术图书的特点，打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持：

善待作者——我们会把出版流程整理得清晰简明，为作者提供优厚的稿酬服务，解除作者的顾虑，安心写作，展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯，并会参照作者实际的工作、生活节奏，量身制定写作计划，确保合作顺利进行。

提升作者——我们打造精品图书，更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力，为作者的事业开拓带来更多的机会。

联系我们

博文视点官网：http://www.broadview.com.cn　　CSDN官方博客：http://blog.csdn.net/broadview2006/

投稿电话：010-51260888　88254368　　投稿邮箱：jsj@phei.com.cn

新浪微博 weibo.com　@博文视点Broadview

微信公众账号　博文视点Broadview

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

关于作者

张子怡（hippo_san），独立设计师、开发者。个人作品遍布于界面交互设计、插画设计、Logo设计、iOS开发、网站开发等领域。同时，也在进行多平台的游戏制作。曾经主导和参与了熊来网、豆瓣FM等优秀产品的制作过程，并创办个人工作室“自然制作”。他相信科技为个人灵魂提供了新的表达方式，并致力于用这种方式来表达自己，同时，解决人类原始、根本的情感需求。

本书特色

本书并不流于说教，而是满满的干货，是 iOS 应用制作的手边书。它打破了设计师和开发者的角色界线，通过实际案例，从软件介绍开始，配合图示，贯穿 iOS 设计、开发领域各种疑难知识，一步一步带领你完成最终上架 App Store 的应用。

上架建议：移动设计/移动开发

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

责任编辑：黄爱萍
封面设计：李　玲

ISBN 978-7-121-32019-4

定价：69.00元